Neural Task Synthesis for Visual Programming

Victor-Alexandru Pădurean
MPI-INF, Germany
vpadurea@mpi-inf.mpg.de

Georgios Tzannetos
MPI-SWS, Germany
gtzannet@mpi-sws.org

Adish Singla
MPI-SWS, Germany
adishs@mpi-sws.org

Abstract

Generative neural models hold great promise in enhancing programming education by synthesizing new content. We seek to design neural models that can automatically generate programming tasks for a given specification in the context of visual programming domains. Despite the recent successes of large generative models like GPT-4, our initial results show that these models are ineffective in synthesizing visual programming tasks and struggle with logical and spatial reasoning. We propose a novel neuro-symbolic technique, NEURTASKSYN, that can synthesize programming tasks for a specification given in the form of desired programming concepts exercised by its solution code and constraints on the visual task. NEURTASKSYN has two components: the first component is trained via imitation learning procedure to generate possible solution codes, and the second component is trained via reinforcement learning procedure to guide an underlying symbolic execution engine that generates visual tasks for these codes. We demonstrate the effectiveness of NEURTASKSYN through an extensive empirical evaluation and a qualitative study on reference tasks taken from the *Hour of Code: Classic Maze* challenge by Code.org and the *Intro to Programming with Karel* course by CodeHS.com.

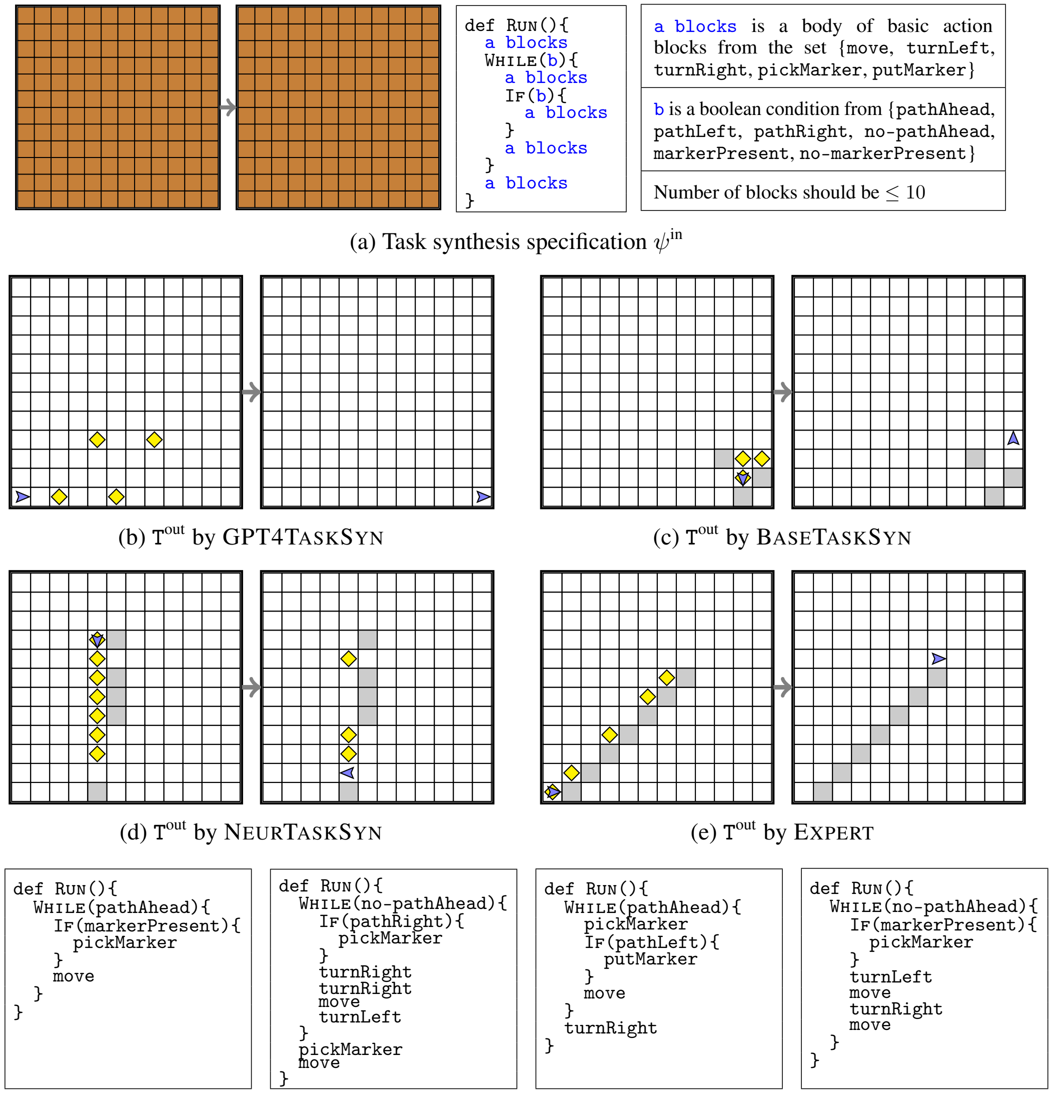
1 Introduction

Recent advances in generative AI have demonstrated impressive performance in a variety of domains, including visual art and music creation [1, 2, 3, 4, 5], medicinal chemistry synthesis [6, 7, 8, 9], and AI-enhanced programming [10, 11]. These successes are, in part, driven by advanced capabilities of deep generative models, such as Stable Diffusion [5], chatGPT [12], and GPT-4 [13]. These advancements also hold great promise in enhancing education, for instance, by generating personalized content and new practice tasks for students allowing them to master required concepts [14, 15, 16, 17].

In this paper, we explore the role of generative AI in visual programming domains used for introductory programming education. Popular domains, such as Scratch [18], *Hour of Code:Maze Challenge* by Code.org (HoCMaze) [19, 20], and Karel [21], have become an integral part of introductory computer science education and are used by millions of students, including children and K-12 students [20, 22, 23]. In existing visual programming platforms, programming tasks are hand-curated by tutors and the available set of tasks is typically very limited, posing a major hurdle for novices in mastering the missing concepts [24, 25]. To this end, we seek to design generative models that can automatically synthesize visual programming tasks for a given specification (e.g., see Figure 1a).

As a natural approach, one might be tempted to employ state-of-the-art models like GPT-4 to generate a visual programming task by providing task synthesis specification as a prompt. In particular, models like GPT-4 are trained on multi-modal data including text, code, and visual data, and hence it seems suitable technique for visual programming domains [13, 28]. However, our initial results show that these models are ineffective in synthesizing visual programming tasks and struggle with logical and spatial reasoning, as has been indicated in recent literature on state-of-the-art models [29, 28, 30, 31]. For instance, GPT-4's $\mathrm{T}^{\mathrm{out}}$ in Figure 1b are not solvable by codes that would match the input specification; see detailed discussion and results in Section 4. In general, a major challenge in using

NeurIPS'23 Workshop on Generative AI for Education (GAIED).



(a) Task synthesis specification ψ^{in}

(b) $\mathtt{T}^{\text{out}}$ by GPT4TaskSyn (c) $\mathtt{T}^{\text{out}}$ by BaseTaskSyn

(d) $\mathtt{T}^{\text{out}}$ by NeurTaskSyn (e) $\mathtt{T}^{\text{out}}$ by Expert

(f) $\mathtt{C}^{\text{out}}$ by GPT4TaskSyn (g) $\mathtt{C}^{\text{out}}$ by BaseTaskSyn (h) $\mathtt{C}^{\text{out}}$ by NeurTaskSyn (i) $\mathtt{C}^{\text{out}}$ by Expert

Figure 1: Illustrative example showcasing task synthesis inspired by the Stairway Karel task [21, 26, 27]. **(a)** Task synthesis specification $\psi^{\text{in}} := (\psi^{\text{in}}_{\text{puzzle}}, \psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta}, \psi^{\text{in}}_{\text{size}})$ is provided as input: $\psi^{\text{in}}_{\text{puzzle}}$ is a single pregrid-postgrid pair with size 12x12 without any initialization of puzzle elements; $\psi^{\text{in}}_{\text{sketch}}$ along with $\psi^{\text{in}}_{\Delta}$ and $\psi^{\text{in}}_{\text{size}} = 10$ specify constraints on code solutions of a synthesized task. **(b–d)** show tasks $\mathtt{T}^{\text{out}}$ by three techniques and **(e)** shows task $\mathtt{T}^{\text{out}}$ based on Stairway. **(f–i)** show codes $\mathtt{C}^{\text{out}}$ used as intermediate step to generate output tasks. See Section 2 and Section 4.

purely neural generative models for synthesizing visual programming tasks is that the generative process is highly brittle – even a small modification in the output task could make it invalid or semantically incorrect w.r.t. the input specification [25].

As an alternate to neural generative models, we could rely on symbolic generative methods driven by search and planning to generate content that matches a specification. Several works have shown the efficacy of symbolic methods to generate new tasks in various educational domains, e.g., algebra exercises [32, 33], geometric proof problems [34], natural deduction [35], mathematical word problems [36], Sokoban puzzles [37], and visual programming tasks [25, 38]. In particular, our work is related to [25, 38] that proposed symbolic methods guided by hand-crafted constraints and Monte Carlo Tree Search to generate high-quality visual programming tasks. However, their symbolic methods still suffer from intractably large spaces of feasible tasks/codes for a given specification, and could take several minutes to generate an output task for an input specification as shown in Figure 1a.

In general, a major shortcoming of using purely symbolic generative methods in the above-mentioned works is that the generative process is typically time-inefficient and not suitable for applications that require online or large-scale synthesis.

Against that backdrop, the main research question is: *Can we develop neuro-symbolic techniques that can synthesize high-quality visual programming tasks while being robust and efficient?* To this end, we develop NEURTASKSYN, a novel neuro-symbolic technique that can synthesize programming tasks for input specifications in the form of desired programming concepts exercised by its solution code and constraints on the visual task. Given a task synthesis specification as input (Figure 1a), NEURTASKSYN uses two components trained via reinforcement learning procedure: the first component generates possible solution codes (Figure 1h), and the second component guides an underlying symbolic execution engine that generates visual tasks for these codes (Figure 1d). Our main results and contributions are summarized below: I. We formalize synthesizing visual programming tasks for a given specification. (Section 2) II. We propose NEURTASKSYN, a novel neuro-symbolic technique for synthesizing visual programming tasks. (Section 3) III. We demonstrate the effectiveness of NEURTASKSYN through an extensive evaluation on task specifications from real-world programming platforms (Section 4) IV. We will publicly release the implementation and datasets to facilitate future research.

2 Problem Setup

Visual programming tasks. We define a task as a tuple $\mathtt{T} := (\mathtt{T}_{\text{puzzle}}, \mathtt{T}_{\text{store}}, \mathtt{T}_{\text{size}})$, where $\mathtt{T}_{\text{puzzle}}$ denotes the visual puzzle, $\mathtt{T}_{\text{store}}$ the available blocks/commands, and $\mathtt{T}_{\text{size}}$ the maximum number of blocks/commands allowed in a solution code. This task space is inspired by popular visual programming domains, including block-based programming domain of *Hour of Code:Maze Challenge* by Code.org (HoCMaze) [19, 20] and text-based programming domain of Karel [21]. For instance, the puzzle $\mathtt{T}_{\text{puzzle}}$ in Figure 1e is based on the STAIRWAY task from *Intro to Programming with Karel* course by *CodeHS.com* [26, 27]; a solution code when executed should transform this single pregrid to its postgrid with $\mathtt{T}_{\text{store}} = \{$`move`, `turnLeft`, `turnRight`, `putMarker`, `pickMarker`, `WHILE`, `IF`$\}$ and $\mathtt{T}_{\text{size}} = 8$.

Code space and solution codes of a task. We define the space of all possible codes in a domain via a domain-specific language (DSL). For instance, in our evaluation with HoCMaze and Karel programming domains, we will use their corresponding DSLs as introduced in [39, 25]. In our code representation, we will indicate specific tokens, including programming constructs and commands in the domain, jointly as "blocks". A code $\mathtt{C}$ has the following attributes: $\mathtt{C}_{\text{blocks}}$ is the set of unique block types in $\mathtt{C}$, $\mathtt{C}_{\text{nblock}}$ is the total number of blocks, $\mathtt{C}_{\text{struct}}$ is the nesting structure of blocks corresponding to programming constructs like loops/conditions, $\mathtt{C}_{\text{depth}}$ is the depth of the corresponding Abstract Syntax Tree (AST), and $\mathtt{C}_{\text{nconst}}$ is the total number of programming constructs. For instance, considering the code $\mathtt{C}$ in Figure 1i, $\mathtt{C}_{\text{blocks}} = \{$`move`, `turnLeft`, `turnRight`, `pickMarker`, `WHILE`, `IF`$\}$, $\mathtt{C}_{\text{nblock}} = 8$, $\mathtt{C}_{\text{struct}} = \{$`RUN` $\{$`WHILE`$\{$`IF`$\}\}\}$, $\mathtt{C}_{\text{depth}} = 3$, and $\mathtt{C}_{\text{nconst}} = 2$. For a given task $\mathtt{T}$, a code $\mathtt{C}$ is a solution code if the following holds: $\mathtt{C}$ successfully solves $\mathtt{T}_{\text{puzzle}}$, $\mathtt{C}_{\text{blocks}} \subseteq \mathtt{T}_{\text{store}}$, and $\mathtt{C}_{\text{nblock}} \leq \mathtt{T}_{\text{size}}$.

Task synthesis specification. We now introduce a notation to specify desired tasks for synthesis that exercise certain programming concepts in their solution codes and respect certain constraints on the visual puzzle. We define a task synthesis specification as a tuple $\psi := (\psi_{\text{puzzle}}, \psi_{\text{sketch}}, \psi_{\Delta}, \psi_{\text{size}})$, where ψ_{puzzle} is partially initialized visual puzzle, ψ_{sketch} is a code sketch (i.e., a partial code) capturing the structure that should be followed by the synthesized task's solution codes along with additional constraints specified by ψ_{Δ} and ψ_{size}. For instance, the task synthesis specification ψ in Figure 1a is inspired by the STAIRWAY Karel task – here, ψ_{puzzle} is a single pregrid-postgrid pair with size 12x12 without any initialization of puzzle elements; ψ_{sketch} along with ψ_{Δ} and $\psi_{\text{size}} = 10$ specify constraints.

Synthesis objective. Given a task synthesis specification $\psi^{\text{in}} := (\psi^{\text{in}}_{\text{puzzle}}, \psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta}, \psi^{\text{in}}_{\text{size}})$ as input, we seek to generate a task $\mathtt{T}^{\text{out}} := (\mathtt{T}^{\text{out}}_{\text{puzzle}}, \mathtt{T}^{\text{out}}_{\text{store}}, \mathtt{T}^{\text{out}}_{\text{size}})$ as output. To formally set our synthesis objective and evaluation metrics, below we introduce different criteria that we want $\mathtt{T}^{\text{out}}$ to satisfy w.r.t. ψ^{in}:

- **O1:Validity.** $\mathtt{T}^{\text{out}}$ is valid w.r.t. ψ^{in} if $\mathtt{T}^{\text{out}}_{\text{puzzle}}$ respects $\psi^{\text{in}}_{\text{puzzle}}$, $\mathtt{T}^{\text{out}}_{\text{store}}$ only contain blocks as allowed by $(\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta})$, and $\mathtt{T}^{\text{out}}_{\text{size}} \leq \psi^{\text{in}}_{\text{size}}$.
- **O2:Solvability.** $\mathtt{T}^{\text{out}}$ is solvable, i.e., there exists at least one solution code.

- **O3:Concepts.** $\texttt{T}^{\text{out}}$ conceptually captures ψ^{in} in the following sense: (a) there exists at least one solution code $\texttt{C}$ for $\texttt{T}^{\text{out}}$ that respects $(\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta})$; (b) any solution code $\texttt{C}$ for $\texttt{T}^{\text{out}}$ has $\texttt{C}_{\text{depth}}$ and $\texttt{C}_{\text{nconst}}$ at least as that required by $\psi^{\text{in}}_{\text{sketch}}$.

In addition, it is desirable that synthesized tasks meet the following properties of real-world tasks:

- **O4:Trace.** For any solution code $\texttt{C}$ for $\texttt{T}^{\text{out}}$ that respects $(\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta})$, the execution trace of $\texttt{C}$ on $\texttt{T}^{\text{out}}$ executes each loop or conditional at least n times. This property is inspired by real-world tasks which are easy to comprehend; we will use $n = 2$ in Section 4 evaluation.
- **O5:Minimality.** For any solution code $\texttt{C}$ for $\texttt{T}^{\text{out}}$ that respects $(\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta})$, it holds that $\texttt{C}_{\text{nblock}} \geq \texttt{T}^{\text{out}}_{\text{size}} - n$. This property is inspired by real-world tasks which ensure that $\texttt{T}^{\text{out}}_{\text{size}}$ is set tightly; we will use $n = 1$ in Section 4 evaluation.

In order to capture these quality criteria into one objective, we include an overall quality metric, denoted as **Overall**. More concretely, for a given specification, **Overall** is 1 if all the synthesis objectives O1–O5 are 1. This metric is a indicator of the overall quality of a generated task in terms of matching conceptual specification (O1, O2, O3) and human-centered aspects of visual quality and suitability (O4, O5).

3 Our Synthesis Technique NeurTaskSyn

In this section, we present NeurTaskSyn, our neuro-symbolic technique to synthesize visual programming tasks ($\texttt{T}^{\text{out}}$) for an input specification (ψ^{in}). We provide an overview of our technique here, and full details are in the supplementary material.

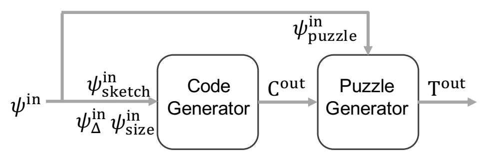


Figure 2: Components of task synthesis.

As noted in Section 1, a key challenge in synthesizing tasks is that the mapping from the space of visual tasks to their solution codes is highly discontinuous – a small modification in the output task could make it invalid or semantically incorrect w.r.t. the input specification [25]. One way to tackle this challenge is to first reason about a possible solution code and then generate visual puzzles based on execution traces of this code [33, 37, 25, 40]. This motivates two components in our synthesis process shown in Figure 2: the first component generates possible solution codes $\texttt{C}^{\text{out}}$ (akin to that of program synthesis [41]); the second component generates visual puzzles for these codes via symbolic execution (akin to the idea of test-case generation [42]). Next, we discuss these components of NeurTaskSyn.

3.1 Generating the Solution Code $\texttt{C}^{\text{out}}$

The code generator component takes elements of the specification, $(\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta}, \psi^{\text{in}}_{\text{size}})$, that enforce constraints on solution codes of the desired task and accordingly generates a possible solution code $\texttt{C}^{\text{out}}$. We first describe a base symbolic engine to generate syntactically valid codes from specifications via random search and then describe a neural model to guide this base engine.

Base symbolic engine. The base engine operates on the AST representation of code sketches (i.e., partial codes) as introduced in Section 2. The engine generates codes by sampling tokens (i.e., blocks, conditions, and iterators) from the underlying DSL while respecting specification. Even though this engine ensures that a generated code is syntactically correct and valid w.r.t. specification, it could have semantic irregularities.

Neural model. The neural model is trained to guide the sampling process of the base symbolic engine. This neural model is akin to a program synthesizer and one could use a variety of architectures, for instance, transformer-based [43, 44, 45, 46] or custom-made encoder-decoder approaches [47, 39, 48]. Our neural architecture is based on the work of [49, 39] that use an LSTM-based decoder [50] for generating program solution for an input visual programming task; in our setting the input corresponds to the specification. Similar to [39], we use imitation (supervised) learning approach to train the neural model. Full implementation and training details are provided in the supplementary material.

3.2 Generating the Visual Puzzle $\texttt{T}^{\text{out}}_{\text{puzzle}}$ and Task $\texttt{T}^{\text{out}}$

The puzzle generator component takes elements of the specification, $\psi^{\text{in}}_{\text{puzzle}}$, that enforce constraints on visual puzzle along with generated code $\texttt{C}^{\text{out}}$ and accordingly generates a visual puzzle $\texttt{T}^{\text{out}}_{\text{puzzle}}$. We first describe a base symbolic engine that performs symbolic execution of $\texttt{C}^{\text{out}}$ to generate semantically valid puzzles via random search and then describe a neural model to guide this base engine.

Base symbolic engine. The base engine performs symbolic execution of $\texttt{C}^{\text{out}}$ on $\psi^{\text{in}}_{\text{puzzle}}$ which has uninitialized elements/unknowns (see Figure 1a and Section 2). This symbolic execution emulates an execution trace of $\texttt{C}^{\text{out}}$ and makes decisions about these unknowns resulting in a concrete instantiation of $\psi^{\text{in}}_{\text{puzzle}}$ to $\texttt{T}^{\text{out}}_{\text{puzzle}}$. The outcome of these decisions affect the quality of the generated $\texttt{T}^{\text{out}}_{\text{puzzle}}$, e.g., the number of times each branch for the code in Figure 1h gets executed would affect the visual quality of the puzzle in Figure 1d. In fact, a code could have potentially unbounded number of possible execution traces and randomly taking decisions would typically lead to a lower quality task [42, 25].

Neural model. The neural model is trained to guide the decision-making process of the base symbolic engine. This neural model can be thought of a reinforcement learning (RL) agent [51] whose goal is to make decisions about the unknowns encountered when symbolically executing a code with the objective of generating high-quality puzzles. Existing works have investigated the use of Monte Carlo Tree Search (MCTS) [52] strategy to guide the symbolic execution for generating better puzzles with fewer resources [37, 25]. However, these works used MCTS at inference time without any learnt policy and could take several minutes to generate an output task for an input specification. To speed up the generation process at inference, we train an RL agent whose reward is defined via a scoring function $\mathcal{F}_{\text{score}}$ that captures the quality of the generated visual puzzle for an input specification; this scoring function is similar in spirit to that used for MCTS in [37, 25]. More concretely, we consider an episodic Markov Decision Process where an episode corresponds to a full symbolic execution, the states capture the status of incomplete puzzle and code execution trace, actions correspond to the decisions needed by symbolic engine to instantiate encountered unknowns, transitions are deterministic, and reward is provided at the end of an episode based on $\mathcal{F}_{\text{score}}(\texttt{T}^{\text{out}}, \texttt{C}^{\text{out}})$. We will instantiate $\mathcal{F}_{\text{score}}$ for a given programming domain. Our neural architecture, inspired by work on program synthesis for visual programming tasks [39, 53], uses a CNN-based encoder for incomplete visual puzzles and combines it with features capturing code execution statistics (e.g., coverage, currently executed code block). We use actor-critic policy gradient method for agent training [51]. Full implementation and training details are provided in the supplementary material.

Outputting the task $\texttt{T}^{\text{out}}$. Finally, we instantiate elements of $\texttt{T}^{\text{out}} := (\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{T}^{\text{out}}_{\text{store}}, \texttt{T}^{\text{out}}_{\text{size}})$. $\texttt{T}^{\text{out}}_{\text{puzzle}}$ is based on the generated puzzle, $\texttt{T}^{\text{out}}_{\text{store}}$ is set to blocks allowed by $(\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta})$, and $\texttt{T}^{\text{out}}_{\text{size}} = \texttt{C}^{\text{out}}_{\text{nblock}}$.[1]

4 Experimental Evaluation with Real-World Task Specifications

In this section, we evaluate our task synthesis technique NEURTASKSYN on real-world specifications. Note that we have trained and evaluated the technique on a synthetic dataset, as described in the supplementary material. For brevity, we present only the evaluation on real-world specifications.

Real-world task specifications. We use a set of 10 task specifications from HoCMaze and Karel domains, shown in Figure 3. These task specifications are inspired by their source tasks (see "Source" column in the figure) in the following sense: we create a specification ψ^{in} for which the corresponding source task is a desired task as would be created by experts. Figure 1 shows illustration of task synthesis for a variant of $\psi 8$ (source as STAIRWAY Karel task) where we used 12x12 grid size.

Techniques evaluated. We evaluate NEURTASKSYN$_{\text{c:10,p:100}}$ with $\text{c} = 10$ and $\text{p} = 100$, i.e., total of $\text{c} \times \text{p} = 1000$ rollouts Next, we describe three additional techniques evaluated:

- BASETASKSYN$_{\text{c:10,p:100}}$ operates similar to NEURTASKSYN$_{\text{c:10,p:100}}$, but uses only base symbolic engine with random search without any neural guidance.
- EXPERT technique simply outputs a task $\texttt{T}^{\text{out}}$ based on the source task associated with input specification ψ^{in}; moreover it appropriately adjusts $\texttt{T}^{\text{out}}_{\text{puzzle}}$ to match $\psi^{\text{in}}_{\text{puzzle}}$ layout, sets $\texttt{T}^{\text{out}}_{\text{store}}$ to blocks as allowed by $(\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta})$, and sets $\texttt{T}^{\text{out}}_{\text{size}}$ as size of the minimal solution code.
- GPT4TASKSYN is technique based on OpenAI's GPT-4, a state-of-the-art large language model [13]. We provide a brief overview of how we use GPT-4 for task synthesis and defer the full details, including prompts and examples, to the supplementary material. We tried several different strategies and prompts to make GPT-4 work for synthesising visual programming tasks – here we report on the strategy based on a two-stage task synthesis process as shown in Figure 2. More concretely, we first ask GPT-4 to generate a code $\texttt{C}^{\text{out}}$ for ψ^{in} and then ask it to generate a puzzle $\texttt{T}^{\text{out}}_{\text{puzzle}}$ that could be solved by $\texttt{C}^{\text{out}}$. The first stage comprised 5 separate queries to generate a $\texttt{C}^{\text{out}}$:

[1]It is possible that the code $\texttt{C}^{\text{out}}$ generated during intermediate step turns out not to be a solution for $\texttt{T}^{\text{out}}$, e.g., when $\texttt{C}^{\text{out}}$ is semantically incorrect and cannot generate a corresponding puzzle. In this case, we set $\texttt{T}^{\text{out}}_{\text{size}} = \psi^{\text{in}}_{\text{size}}$.

ψ^{in}	$\psi^{\text{in}}_{\text{sketch}}$ structure	(depth, constructs)	$\psi^{\text{in}}_{\text{puzzle}}$	$\psi^{\text{in}}_{\Delta}$ and $\psi^{\text{in}}_{\text{size}}$	Source
$\psi 0$	{RUN {REPEAT}}	$(2,1)$	16x16 empty	HoCMaze, blocks ≤ 10	HoC:Maze9 [19]
$\psi 1$	{RUN {REPEATUNTIL}}	$(2,1)$	16x16 empty	HoCMaze, blocks ≤ 10	HoC:Maze13 [19]
$\psi 2$	{RUN {REPEAT; REPEAT}}	$(2,2)$	16x16 empty	HoCMaze, blocks ≤ 10	HoC:Maze8 [19]
$\psi 3$	{RUN {REPEATUNTIL{IFELSE}}}	$(3,2)$	16x16 empty	HoCMaze, blocks ≤ 10	HoC:Maze18 [19]
$\psi 4$	{RUN {REPEATUNTIL{IF; IF}}}	$(3,3)$	16x16 empty	HoCMaze, blocks ≤ 10	HoC:Maze20 [19]
$\psi 5$	{RUN}	$(1,0)$	16x16 empty	Karel, blocks ≤ 10	Karel:OurFirst [26]
$\psi 6$	{RUN {WHILE}}	$(2,1)$	16x16 empty	Karel, blocks ≤ 10	Karel:Diagonal [26]
$\psi 7$	{RUN {WHILE; WHILE}}	$(2,2)$	16x16 empty	Karel, blocks ≤ 10	Karel:RowBack [26]
$\psi 8$	{RUN {WHILE{IF}}}	$(3,2)$	16x16 empty	Karel, blocks ≤ 10	Karel:Stairway [26]
$\psi 9$	{RUN {WHILE{REPEAT}}}	$(3,2)$	16x16 empty	Karel, blocks ≤ 10	Karel:CleanAll

Figure 3: Real-world task specifications for HoCMaze and Karel; $\psi^{\text{in}}_{\text{sketch}}$ is shortened for brevity.

Technique	O1:Validity	O2:Solvability	O3:Concepts	O4:Trace	O5:Minimality	Overall	$\mathtt{C}^{\text{out}}$ solves $\mathtt{T}^{\text{out}}$
NEURTASKSYN$_{\text{c:10,p:100}}$	1.00 (0.00)	1.00 (0.00)	0.83 (0.04)	0.80 (0.00)	0.77 (0.11)	0.73 (0.08)	1.00 (0.00)
BASETASKSYN$_{\text{c:10,p:100}}$	1.00 (0.00)	0.97 (0.04)	0.37 (0.08)	0.33 (0.04)	0.23 (0.04)	0.20 (0.00)	0.50 (0.12)
GPT4TASKSYN	1.00 (0.00)	0.97 (0.04)	0.57 (0.11)	0.60 (0.07)	0.47 (0.08)	0.27 (0.08)	0.33 (0.08)
EXPERT	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Figure 4: Results on real-world task specifications for HoCMaze and Karel in Figure 3; see Section 4.

each query started with an initial prompt and then follow-up prompts to fix any mistakes. The second stage comprised of another 5 separate queries to generate a puzzle $\mathtt{T}^{\text{out}}_{\text{puzzle}}$: each query started with an initial prompt and then follow-up prompts to fix any mistakes. Once we get $\mathtt{C}^{\text{out}}$ and $\mathtt{T}^{\text{out}}_{\text{puzzle}}$, we set other elements of the task, $\mathtt{T}^{\text{out}}_{\text{store}}$ and $\mathtt{T}^{\text{out}}_{\text{size}}$, as for BASETASKSYN and NEURTASKSYN.

Evaluation metrics. We evaluate these techniques w.r.t. six metrics, each corresponding to the synthesis objective introduced in Section 2, numbered O1–O5 and Overall. Additionally, we also report a binary metric of whether $\mathtt{C}^{\text{out}}$ solves $\mathtt{T}^{\text{out}}$ which provide insights into these objectives. Even though these objectives are quantitative, it is challenging to fully automate their evaluation because it requires analyzing properties of different possible solution codes of a generated task. We manually did this evaluation when computing performance for each technique and metric. Results are reported as a mean over 10 specifications ψ^{in} from Figure 3; for NEURTASKSYN$_{\text{c:10,p:100}}$ and BASETASKSYN$_{\text{c:10,p:100}}$, we evaluate over three different seeds and report averaged results as mean (stderr).

Results. Figure 4 reports evaluation results for different techniques w.r.t. our task synthesis objectives. Next, we summarize some of our key findings. First, NEURTASKSYN has high performance of over 0.7 across all metrics. The illustrative example in Figure 1 showcases the high-quality of tasks synthesized by NEURTASKSYN, matching interesting characteristics of real-world tasks from EXPERT. Second, GPT4TASKSYN and BASETASKSYN struggle on objectives O3, O4, and O5. This low performance of GPT4TASKSYN and BASETASKSYN can be explained, in part, by their failure to generate a valid task/code pair. The illustrative examples in Figure 1 further highlights the issues of tasks generated by these techniques. GPT4TASKSYN's $\mathtt{T}^{\text{out}}$ in Figure 1b is not solvable by codes that would match the input specification. In summary, these results highlight the challenges in synthesizing visual programming tasks by state-of-the-art generative models as the synthesis process requires logical, spatial, and programming skills. Moreover, these results demonstrate the effectiveness of NEURTASKSYN in synthesizing high-quality visual programming tasks for real-world specifications.

5 Concluding Discussions

We developed a novel neuro-symbolic technique, NEURTASKSYN, that can synthesize visual programming tasks for a given specification. We demonstrated the effectiveness of NEURTASKSYN through an extensive evaluation on reference tasks from popular visual programming environments. We believe our proposed technique has the potential to drastically enhance introductory programming education by synthesizing personalized content for students. Next, we discuss a few limitations of our current work and outline a plan for future work. First, the two neural components of NEURTASKSYN use LSTM/CNN-based architecture and trained from scratch; it would be interesting to fine-tune models like GPT-4 or (an open-source variant) CodeT5 [46] for synthesizing visual programming tasks. Second, our methodology focused on visual programming; it would be interesting to develop generative models for synthesizing tasks in other programming domains, e.g., Python problems that match a given specification. Third, our evaluation study focused on six objectives but didn't involve human evaluation; in the future, it would be important to conduct user studies with educators and students to evaluate the quality of synthesized tasks.

Acknowledgments and Disclosure of Funding

Funded/Co-funded by the European Union (ERC, TOPS, 101039090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Hao-Wen Dong, Wen-Yi Hsiao, Li-Chia Yang, and Yi-Hsuan Yang. Musegan: Multi-track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment. In *AAAI*, 2018.

[2] Jean-Pierre Briot, Gaëtan Hadjeres, and François-David Pachet. *Deep Learning Techniques for Music Generation*. Springer, 2020.

[3] Minhyang Suh, Emily Youngblom, Michael Terry, and Carrie J. Cai. AI as Social Glue: Uncovering the Roles of Deep Generative AI during Social Music Composition. In *CHI*, 2021.

[4] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-Shot Text-to-Image Generation. In *ICML*, 2021.

[5] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-Resolution Image Synthesis with Latent Diffusion Models. In *CVPR*, 2022.

[6] Petra Schneider, W Patrick Walters, Alleyn T Plowright, Norman Sieroka, Jennifer Listgarten, Robert A Goodnow Jr, Jasmin Fisher, Johanna M Jansen, José S Duca, Thomas S Rush, et al. Rethinking Drug Design in the Artificial Intelligence Era. *Nature Reviews Drug Discovery*, 19(5):353–364, 2020.

[7] W. Walters and Mark Murcko. Assessing the Impact of Generative AI on Medicinal Chemistry. *Nature Biotechnology*, 38(2):143–145, 2020.

[8] Xiaochu Tong, Xiaohong Liu, Xiaoqin Tan, Xutong Li, Jiaxin Jiang, Zhaoping Xiong, Tingyang Xu, Hualiang Jiang, Nan Qiao, and Mingyue Zheng. Generative Models for De Novo Drug Design. *Journal of Medicinal Chemistry*, 64(19):14011–14027, 2021.

[9] Wenhao Gao and Connor W Coley. The Synthesizability of Molecules Proposed by Generative Models. *Journal of Chemical Information and Modeling*, 60(12):5714–5723, 2020.

[10] Mark Chen et al. Evaluating Large Language Models Trained on Code. *CoRR*, abs/2107-03374, 2021.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP*, 2020.

[12] OpenAI. ChatGPT. `https://openai.com/blog/chatgpt`, 2023.

[13] OpenAI. GPT-4 Technical Report. *CoRR*, abs/2303.08774, 2023.

[14] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *ICER*, 2022.

[15] Tamara Tate, Shayan Doroudi, Daniel Ritchie, and Ying Xu. Educational Research and AI-Generated Writing: Confronting the Coming Tsunami. 2023.

[16] David Baidoo-Anu and Leticia Owusu Ansah. Education in the Era of Generative Artificial Intelligence (AI): Understanding the Potential Benefits of ChatGPT in Promoting Teaching and Learning. *Available at SSRN 4337484*, 2023.

[17] Weng Marc Lim, Asanka Gunasekara, Jessica Leigh Pallant, Jason Ian Pallant, and Ekaterina Pechenkina. Generative AI and the Future of Education: Ragnarök or Reformation? A Paradoxical Perspective from Management Educators. *The International Journal of Management Education*, 21(2):100790, 2023.

[18] Mitchel Resnick, John H. Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay S. Silver, Brian Silverman, and Yasmin B. Kafai. Scratch: Programming for All. *Communications of ACM*, 52(11):60–67, 2009.

[19] Code.org. Hour of Code: Classic Maze Challenge. `https://studio.code.org/s/hourofcode`, 2013.

[20] Code.org. Code.org: Learn Computer Science. `https://code.org/`, 2013.

[21] Richard E Pattis, Jim Roberts, and Mark Stehlik. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., 1995.

[22] Mike Wu, Milan Mosse, Noah D. Goodman, and Chris Piech. Zero Shot Learning for Code Education: Rubric Sampling with Deep Learning Inference. In *AAAI*, 2019.

[23] Thomas W. Price and Tiffany Barnes. Position Paper: Block-Based Programming Should Offer Intelligent Support for Learners. *IEEE Blocks and Beyond Workshop*, 2017.

[24] Rui Zhi, Thomas W. Price, Samiha Marwan, Alexandra Milliken, Tiffany Barnes, and Min Chi. Exploring the Impact of Worked Examples in a Novice Programming Environment. In *SIGCSE*, 2019.

[25] Umair Z. Ahmed, Maria Christakis, Aleksandr Efremov, Nigel Fernandez, Ahana Ghosh, Abhik Roychoudhury, and Adish Singla. Synthesizing Tasks for Block-based Programming. In *NeurIPS*, 2020.

[26] CodeHS. Intro to Programming with Karel the Dog. `https://codehs.com/info/curriculum/introkarel`, 2012.

[27] CodeHS. CodeHS.com: Teaching Coding and Computer Science. `https://codehs.com/`, 2012.

[28] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. Sparks of Artificial General Intelligence: Early Experiments with GPT-4. *CoRR*, abs/2303.12712, 2023.

[29] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, Quyet V. Do, Yan Xu, and Pascale Fung. A Multitask, Multilingual, Multimodal Evaluation of Chatgpt on Reasoning, Hallucination, and Interactivity. *CoRR*, abs/2302-04023, 2023.

[30] Karthik Valmeekam, Alberto Olmo Hernandez, Sarath Sreedharan, and Subbarao Kambhampati. Large Language Models Still Can't Plan (A Benchmark for LLMs on Planning and Reasoning about Change). *CoRR*, abs/2206-10498, 2022.

[31] Jie Huang and Kevin Chen-Chuan Chang. Towards Reasoning in Large Language Models: A Survey. *CoRR*, abs/2212-10403, 2022.

[32] Rohit Singh, Sumit Gulwani, and Sriram K. Rajamani. Automatically Generating Algebra Problems. In *AAAI*, 2012.

[33] Sumit Gulwani. Example-based Learning in Computer-aided STEM Education. *Communications of the ACM*, 57(8):70–80, 2014.

[34] Chris Alvin, Sumit Gulwani, Rupak Majumdar, and Supratik Mukhopadhyay. Synthesis of Geometry Proof Problems. In *AAAI*, 2014.

[35] Umair Z. Ahmed, Sumit Gulwani, and Amey Karkare. Automatically Generating Problems and Solutions for Natural deduction. In *IJCAI*, 2013.

[36] Oleksandr Polozov, Eleanor O'Rourke, Adam M. Smith, Luke Zettlemoyer, Sumit Gulwani, and Zoran Popovic. Personalized Mathematical Word Problem Generation. In *IJCAI*, 2015.

[37] Bilal Kartal, Nick Sohre, and Stephen J. Guy. Data Driven Sokoban Puzzle Generation with Monte Carlo Tree Search. In *AIIDE*, 2016.

[38] Ahana Ghosh, Sebastian Tschiatschek, Sam Devlin, and Adish Singla. Adaptive Scaffolding in Block-Based Programming via Synthesizing New Tasks as Pop Quizzes. In *AIED*, 2022.

[39] Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. In *ICLR*, 2018.

[40] Alperen Tercan, Ahana Ghosh, Hasan Ferit Eniser, Maria Christakis, and Adish Singla. Synthesizing a Progression of Subtasks for Block-Based Visual Programming Tasks. *CoRR*, abs/2305.17518, 2023.

[41] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program Synthesis. *Foundations and Trends® in Programming Languages*, 2017.

[42] James C. King. Symbolic Execution and Program Testing. *Communications of ACM*, 19(7):385–394, 1976.

[43] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. Coderl: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In *NeurIPS*, 2022.

[44] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A Generative model for Code Infilling and Synthesis. *CoRR*, abs/2204.05999, 2022.

[45] Yujia Li et al. Competition-Level Code Generation with Alphacode. *CoRR*, abs/2203.07814, 2022.

[46] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *EMNLP*, 2021.

[47] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to Write Programs. In *ICLR*, 2017.

[48] Pengcheng Yin and Graham Neubig. A Syntactic Neural Model for General-Purpose Code Generation. In *ACL*, 2017.

[49] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural Program Learning under Noisy I/O. In *ICML*, 2017.

[50] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9, 1997.

[51] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

[52] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo Planning. In *ECML*, 2006.

[53] Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. Synthesize, Execute and Debug: Learning to Repair for Neural Program Synthesis. In *NeurIPS*, 2020.

[54] Richard Shin, Neel Kant, Kavi Gupta, Chris Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. Synthetic Datasets for Neural Program Synthesis. In *ICLR*, 2019.

[55] Aleksandr Efremov, Ahana Ghosh, and Adish Singla. Zero-shot Learning of Hint Policy via Reinforcement Learning and Program Synthesis. In *EDM*, 2020.

[56] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas J. Guibas. Autonomously Generating Hints by Inferring Problem Solving Policies. In *L@S*, 2015.

[57] Adish Singla and Nikitas Theodoropoulos. From {Solution Synthesis} to {Student Attempt Synthesis} for Block-Based Visual Programming Tasks. In *EDM*, 2022.

A Table of Contents

In this section, we provide a brief description of the content provided in the appendices of the paper.

- Appendix B provides a discussion of the broader impact of our work and compute resources used.
- Appendix C presents the details about the generation of the illustrative example from Figure 1 and shows its relationship with metrics O1-O5 described in Sections 2. Additionally, we introduce an illustrative example for HoCMaze.
- Appendix D gives more insights into the architecture described in Section 3.
- Appendix E presents an evaluation of NEURTASKSYN on a synthetic dataset and offers details about the scoring functions, the synthetic dataset creation process, and the training process.
- Appendix G provides the source task/code pairs used for creating the real-world task specifications in Section 4. It also provides more insights into the interaction with GPT-4.

B Discussion

Broader impact. This paper develops new techniques which have the potential of being used for improving pedagogy in visual programming environments. On the existing platforms, content is hand-curated by tutors, offering limited resources for students to practice on. We aim to tackle this challenge by synthesizing novel practice tasks that match a desired level of difficulty with regard to exercised content for a student. We believe our proposed technique has the potential to drastically enhance introductory programming education by synthesizing personalized content for students.

Compute resources. All the experiments were conducted on a cluster of machines equipped with Intel Xeon Gold 6142 CPUs running at a frequency of 2.60GHz.

C Illustrative Examples: Details

In this section, we discuss the details regarding the generation and scoring for each of the techniques' output in Figure 1 for Karel. We introduce an additional illustrative example and offer details for HoCMaze as well.

C.1 Example for Karel in Figure 1

We present $\mathtt{T}^{\text{out}}$, along with $\mathtt{C}^{\text{out}}$ for each of the techniques with ψ^{in} as input in Figure 6; this figure expands on Figure 1 with additional details. We give additional explanations regarding how each of the techniques' output respects or not metrics O1-O5 (see Sections 2 and 4) in Figure 5.

Generation/adjustment for GPT4TASKSYN in Figure 1. For this example, we set $\psi^{\text{in}}_{\text{puzzle}}$ as an empty 12x12 grid and query GPT-4.

Generation/adjustment for BASETASKSYN and NEURTASKSYN in Figure 1. The neural model for puzzle generation is trained on 16x16 grids, yet the symbolic engine can support the existence of pre-initialized grids. Thus, we mask the upper-left part of the grid (4 rows and 4 columns), obtaining the 12x12 workspace for the technique.

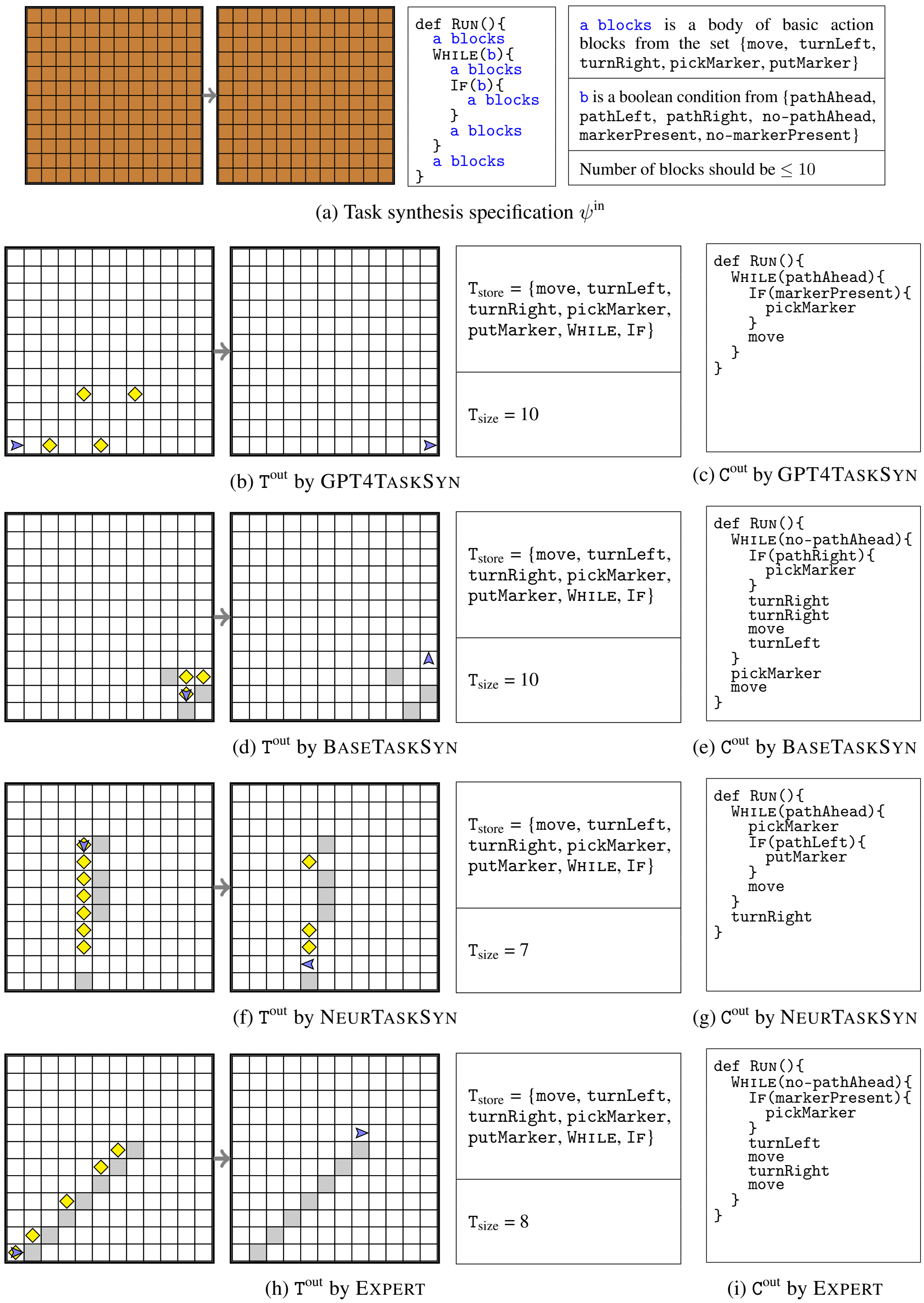
Generation/adjustment for EXPERT in Figure 1. The output of EXPERT for this example is based on the Karel:Stairway task.

Technique	O1:Validity	O2:Solvability	O3:Concepts	O4:Trace	O5:Minimality	Overall	$\mathtt{C}^{\text{out}}$ solves $\mathtt{T}^{\text{out}}$
GPT4TASKSYN	1	0	0	0	0	0	0
BASETASKSYN	1	1	0	1	1	0	1
NEURTASKSYN	1	1	1	1	1	1	1
EXPERT	1	1	1	1	1	1	1

Figure 5: Scores showing whether the output $\mathtt{T}^{\text{out}}$ for ψ^{in} of each technique respects the six metrics O1-O5 and Overall, with the additional $\mathtt{C}^{\text{out}}$ solves $\mathtt{T}^{\text{out}}$ metric, for this Karel example.

We provide explanations for each 0 entry in Figure 5:

- GPT4TASKSYN–O2: $\mathtt{T}^{\text{out}}_{\text{puzzle}}$ cannot be solved with any code respecting $\mathtt{T}^{\text{out}}_{\text{size}}$, hence O2 is 0.
- GPT4TASKSYN–O3: $\mathtt{T}^{\text{out}}_{\text{puzzle}}$ cannot be solved with any code respecting $\mathtt{T}^{\text{out}}_{\text{size}}$, hence O3 is 0.
- GPT4TASKSYN–O4: $\mathtt{T}^{\text{out}}_{\text{puzzle}}$ cannot be solved with any code respecting $\mathtt{T}^{\text{out}}_{\text{size}}$, hence O4 is 0.
- GPT4TASKSYN–O5: $\mathtt{T}^{\text{out}}_{\text{puzzle}}$ cannot be solved with any code respecting $\mathtt{T}^{\text{out}}_{\text{size}}$, hence O5 is 0.
- GPT4TASKSYN–$\mathtt{C}^{\text{out}}$ solves $\mathtt{T}^{\text{out}}$: The generated code $\mathtt{C}^{\text{out}}$ does not solve $\mathtt{T}^{\text{out}}$, i.e., the pregrid is not transformed into the postgrid after code execution.
- BASETASKSYN–O3: The employed `IF` block is not required. This implies that there is a solution code $\mathtt{C}$ which has $\mathtt{C}_{\text{depth}}$ and $\mathtt{C}_{\text{nconst}}$ less than required by $\psi^{\text{in}}_{\text{sketch}}$, hence O3 is 0.



(a) Task synthesis specification ψ^{in}

(b) T^{out} by GPT4TASKSYN

(c) C^{out} by GPT4TASKSYN

(d) T^{out} by BASETASKSYN

(e) C^{out} by BASETASKSYN

(f) T^{out} by NEURTASKSYN

(g) C^{out} by NEURTASKSYN

(h) T^{out} by EXPERT

(i) C^{out} by EXPERT

Figure 6: Illustration containing the tuple T^{out} for each technique, along with C^{out}, for this Karel example. This figure contains all the elements needed in T^{out}, completing Figure 1.

C.2 Example for HoCMaze

We present $\texttt{T}^{\text{out}}$, along with $\texttt{C}^{\text{out}}$ for each of the techniques with ψ^{in} as input in Figure 8. We give additional explanations regarding how each of the techniques' output respects or not metrics O1-O5 (see Sections 2 and 4) in Figure 7.

Generation/adjustment for GPT4TASKSYN. When querying GPT-4 for this example, we set $\psi^{\text{in}}_{\text{puzzle}}$ as an empty 8x8 grid. We then expand the generated grid to a 12x12 grid and manually integrate the pattern seen in Figure 8a to match the specification.

Generation/adjustment for BASETASKSYN and NEURTASKSYN. The neural model for puzzle generation is trained on 16x16 grids, yet the symbolic engine can support the existence of pre-initialized grids. Thus, we mask the upper-left part of the grid (4 rows and 4 columns), obtaining the 12x12 workspace for the technique. On top of that, on the remaining 12x12 grid, we pre-initialize the pattern seen in Figure 8a (i.e., lower-left and upper-right bounded squares).

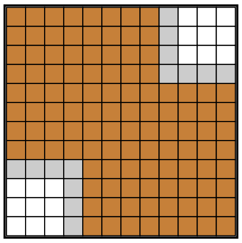
Generation/adjustment for EXPERT. The output of EXPERT represents a manual adaptation of the HoC:Maze18 task to expand it to a 12x12 grid and to integrate the pattern seen in Figure 8a.

Technique	O1:Validity	O2:Solvability	O3:Concepts	O4:Trace	O5:Minimality	Overall	$\texttt{C}^{\text{out}}$ solves $\texttt{T}^{\text{out}}$
GPT4TASKSYN	1	1	0	0	0	0	0
BASETASKSYN	1	1	0	0	0	0	1
NEURTASKSYN	1	1	1	1	1	1	1
EXPERT	1	1	1	1	1	1	1

Figure 7: Scores showing whether the output $\texttt{T}^{\text{out}}$ for ψ^{in} of each technique respects the six metrics O1-O5 and Overall, with the additional $\texttt{C}^{\text{out}}$ solves $\texttt{T}^{\text{out}}$ metric, for this HoCMaze example.

We provide explanations for each 0 entry in Figure 7:

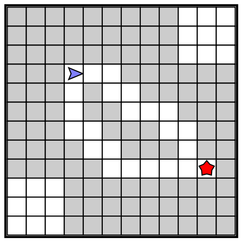
- GPT4TASKSYN–O3: The only possible solution code $\texttt{C}$ has depth 4 and uses 3 constructs (nested IFELSE is needed), i.e., $(\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta})$ is not respected, hence O3 is 0.
- GPT4TASKSYN–O4: There is no solution $\texttt{C}$ that respects $(\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta})$, hence O4 is 0 by definition.
- GPT4TASKSYN–O5: There is no solution $\texttt{C}$ that respects $(\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta})$, hence O5 is 0 by definition.
- GPT4TASKSYN-$\texttt{C}^{\text{out}}$ solves $\texttt{T}^{\text{out}}$: $\texttt{C}^{\text{out}}$, when executed on $\texttt{T}^{\text{out}}_{\text{puzzle}}$, makes the avatar crash into a wall, hence $\texttt{C}^{\text{out}}$ is not a solution for $\texttt{T}^{\text{out}}$
- BASETASKSYN–O3: The `IFELSE` block employed by $\texttt{C}^{\text{out}}$ is not required. This implies that there is a solution code $\texttt{C}$ which has $\texttt{C}_{\text{depth}}$ and $\texttt{C}_{\text{nconst}}$ less than required by $\psi^{\text{in}}_{\text{sketch}}$, hence O3 is 0.
- BASETASKSYN–O4: As the employed `IFELSE` block is not required, it is possible to design a solution code that uses `IFELSE` with a different conditional (e.g., `IF(pathLeft)ELSE`) for which the body would never be executed, hence O4 is 0.
- BASETASKSYN–O5: By making use of the `IFELSE` block (i.e., a possible solution would contain `IF(pathLeft){turnLeft}ELSE{move}`), we can remove the two initial `turnLeft` blocks, thus reducing $\texttt{C}_{\text{nblock}}$ below $(\texttt{T}^{\text{out}}_{\text{size}} - 1)$, hence O5 is 0.



```
def Run(){
  a blocks
  RepeatUntil(goal){
    a blocks
    If(b){
      a blocks
    }
    Else{
      a blocks
    }
    a blocks
  }
}
```

a blocks is a body of basic action blocks from the set {move, turnLeft, turnRight}
b is a boolean condition from {pathAhead, pathLeft, pathRight}
Number of blocks should be ≤ 7

(a) Task synthesis specification ψ^{in}

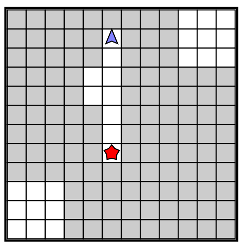


T_{store} = {move, turnLeft, turnRight, RepeatUntil, IfElse}
$T_{\text{size}} = 7$

(b) T^{out} by GPT4TaskSyn

```
def Run(){
  RepeatUntil(goal){
    If(pathLeft){
      turnLeft
      move
    }
    Else{
      move
    }
  }
}
```

(c) C^{out} by GPT4TaskSyn

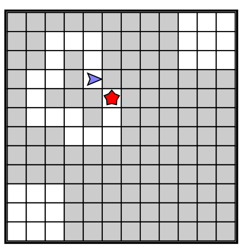


T_{store} = {move, turnLeft, turnRight, RepeatUntil, IfElse}
$T_{\text{size}} = 7$

(d) T^{out} by BaseTaskSyn

```
def Run(){
  turnLeft
  turnLeft
  RepeatUntil(goal){
    If(pathRight){
      move
    }
    Else{
      move
    }
  }
}
```

(e) C^{out} by BaseTaskSyn

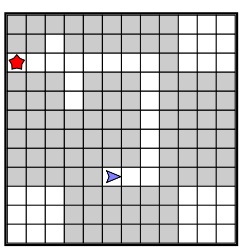


T_{store} = {move, turnLeft, turnRight, RepeatUntil, IfElse}
$T_{\text{size}} = 7$

(f) T^{out} by NeurTaskSyn

```
def Run(){
  RepeatUntil(goal){
    If(pathRight){
      turnRight
    }
    Else{
      turnLeft
      move
    }
    move
  }
}
```

(g) C^{out} by NeurTaskSyn



T_{store} = {move, turnLeft, turnRight, RepeatUntil, IfElse}
$T_{\text{size}} = 5$

(h) T^{out} by Expert

```
def Run(){
  RepeatUntil(goal){
    If(pathAhead){
      move
    }
    Else{
      turnLeft
    }
  }
}
```

(i) C^{out} by Expert

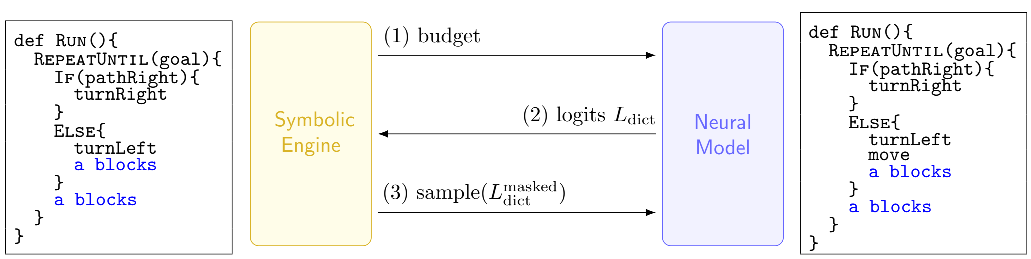
Figure 8: Illustrative example showcasing task synthesis inspired by the Maze18 HoCMaze task [19, 20]. **(a)** Task synthesis specification $\psi^{\text{in}} := (\psi^{\text{in}}_{\text{puzzle}}, \psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta}, \psi^{\text{in}}_{\text{size}})$ is provided as input: $\psi^{\text{in}}_{\text{puzzle}}$ is a 12x12 maze with certain cells initialized to free (white) or wall (gray) cells; $\psi^{\text{in}}_{\text{sketch}}$ along with $\psi^{\text{in}}_{\Delta}$ and $\psi^{\text{in}}_{\text{size}} = 7$ specify constraints on code solutions of a synthesized task. **(b–d)** show tasks T^{out} by three synthesis techniques and **(e)** shows task T^{out} based on Maze18. **(f–i)** show codes C^{out} used as an intermediate step to generate output tasks. See Sections 2 and 4.

D Our Synthesis Technique NEURTASKSYN: Details

Next, we give additional details regarding each module of our architecture. We present the interaction between the neural models and the underlying symbolic engines, the neural architecture, and the training procedures.

D.1 Generating the Solution Code $\mathtt{C}^{\text{out}}$

Code generator visualization. We describe the interaction between the neural model and the underlying symbolic engine for the code generator. For better understanding, we use one concrete example, illustrated in Figure 9. We consider the AST at time t as presented in Figure 9a, where the previously taken decision was the addition of the `turnLeft` token. The symbolic engine continues its depth-first traversal of the AST and now needs to take the next decision for the subsequent '`a blocks`'. This is achieved by interrogating the neural component; interaction demonstrated in Figure 9b. We introduce the notion of a *budget*, which represents the number of available blocks that can be added to the AST so that $\psi^{\text{in}}_{\text{size}}$ is respected; in our example, the remaining budget is 2. It is passed as input for the neural model at time t. The neural model, based on the budget and its internal state, which keeps track of the previously taken decisions, outputs a logit for each decision, i.e., a set of logits L_{dict}. The symbolic engine accepts L_{dict} and masks them according to the rules in the DSL, thus obtaining $L^{\text{masked}}_{\text{dict}}$. In our example, the only values in $L^{\text{masked}}_{\text{dict}}$ that are not masked are those of basic action blocks (i.e., `move`, `turnLeft`, `turnRight`) and the token that represents the end of the ELSE body. After mapping the logits to a probability distribution, the symbolic engine proceeds to sample a decision from it. In our example, `move` is sampled. The decision is passed to the neural model to update its internal state. The symbolic engine then updates the AST with the taken decision (i.e., `move`), thus obtaining the updated version of the AST for the next step at time $t+1$, illustrated in Figure 9c. We generalize this process to every decision that needs to be taken while traversing the AST.



(a) Code status at time t (b) Interaction at time t (c) Code status at time $t+1$

Figure 9: Visualization of the interaction process between the neural model and the symbolic engine in the code generator component of NEURTASKSYN. (a) shows the AST at time t, where the first '`a blocks`' needs to be decided. (b) shows the interaction between the symbolic engine and the neural model at time t, where the symbolic engine first passes the *budget* (available blocks) to the neural model, the neural model computes the logits for all the tokens in the dictionary L_{dict} and passes them back to the symbolic engine, which finally masks them obtaining $L^{\text{masked}}_{\text{dict}}$, applies softmax to obtain a probability and samples the next action, sending it to the neural model. (c) shows the AST at time $t+1$, where the sampled `move` was integrated.

Imitation learning procedure. We will now give details about the learning procedure we used for the neural model. Given the fact that dataset $\mathbb{D} := \{\psi^{\text{in}}\}$ is accompanied by example codes, i.e., $\mathtt{C}^{\text{in}}$ for each ψ^{in}, we employ an imitation (supervised) learning approach, similar to [49, 39]. Thus, for each decision, we compute the cross-entropy with respect to the target decision. We force the agent to take the target decision afterward so the generated code does not digress from our example code.

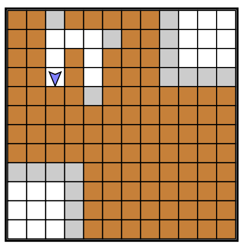
Neural architecture. Here, we present in detail the architecture of the neural model we employ for code generation. Similar to [39], we employ an LSTM-based [50] recurrent neural network. We first convert code tokens to indexes based on a dictionary, then we pass them through an embedding layer. We do the same with the numeric representation of the *budget* (introduced previously). We concatenate both embeddings and pass them through a two-layer LSTM. Last, we convert the output of the LSTM to logits for each entry in the dictionary using a linear layer. The architecture can be observed in Figure 10.

Input	Code token categorical (0-58)	Budget ordinal (0-16)
Embedding	Size = 256	Size = 16
LSTM 1	Hidden dim = 256	
LSTM 2	Hidden dim = 256	
Linear	Hidden dim $\times$ Dict size = 256 $\times$ 59	

Figure 10: Architecture of the neural model used by the code generator.

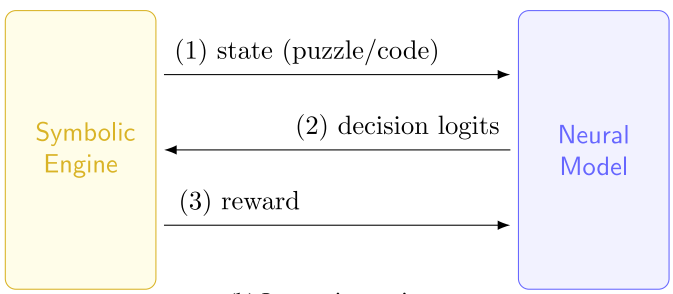
D.2 Generating the Visual Puzzle $\mathtt{T}^{\text{out}}_{\text{puzzle}}$

Puzzle generator visualization. We describe the interaction between the neural model and the underlying symbolic engine for the puzzle generator. We will use a concrete example, visualized in Figure 11. We use an emulator specific to each DSL to execute $\mathtt{C}^{\text{out}}$. Let us consider that the interaction led to the state at time t presented in Figure 11a. Here, the code execution status is represented by the emulator doing the `IF(pathRight)` interrogation upon the symbolic executor, and the visual puzzle status represents the avatar with an *unknown* to its right. This requires an interaction between the symbolic engine and the neural model, as depicted in Figure 11b. The neural model receives the current state of the symbolic engine (i.e., puzzle status and code execution status, as in Figure 11a) and outputs the logits for each possible decision (e.g., path to the right or not). The symbolic engine maps the logits to a probability distribution and samples a decision. In our case, the decision is that there is a path to the right. It then executes the upcoming blocks (i.e., `turnRight`, `move`) until reaching a new decision point, i.e., `REPEATUNTIL (goal)`. It also computes the reward (as we explain in the next subsection) and passes it back to the neural model. The state at time $t + 1$ is given by the new visual puzzle status and the code execution status, which has reached a new decision point `REPEATUNTIL (goal)`, as depicted in Figure 11c. This process can be generalized for every decision and the location/orientation initialization.

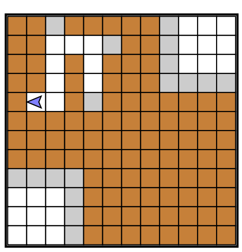


```
def RUN(){
  REPEATUNTIL(goal){
    IF(pathRight){
      turnRight
    }
    ELSE{
      turnLeft
      move
    }
    move
  }
}
```

(a) State given by puzzle status and code execution status at time t



(b) Interaction at time t



```
def RUN(){
  REPEATUNTIL(goal){
    IF(pathRight){
      turnRight
    }
    ELSE{
      turnLeft
      move
    }
    move
  }
}
```

(c) State given by puzzle status and code execution status at time $t + 1$

Figure 11: Visualization of the interaction process between neural model and the symbolic engine in the puzzle generator component of NEURTASKSYN. (a) shows the state at time t, comprised of the visual aspect of the puzzle and the code execution status. (b) shows the interaction between the symbolic engine and the neural model at time t, where the symbolic engine first passes the state to the neural model, the neural model outputs the logits for the decision, and then, the symbolic engine, after sampling and executing a decision, offers a reward to the neural model. (c) shows the new state at time $t + 1$, comprised of the new visual aspect of the puzzle and the updated code execution status.

Reinforcement learning procedure. We describe the details necessary for training our neural model for puzzle generation as a reinforcement learning (RL) agent. In the usual RL setting, the agent interacts with an environment, modelled as a Markov Decision Process (MDP) [51]. The MDP is a tuple $M = (S, A, P, R, S_0)$, where:

- S is the set of possible states s. A state s is given by the current puzzle status, the current code execution status, and the current code trace (see Figures 11a and 11c);
- $A = \cup_{s \in S} A_s$ is the set of all possible decisions, and A_s is the set of decisions possible in state s;
- $P : S \times A \times S \to \mathbb{R}$ denotes the transition dynamics. $P(s'|s, a)$ is defined only for $a \in A_s$. We have $P(s'|s, a) = 1$ for $s' = s \oplus a$, and 0 otherwise;
- $R : S \times A \to \mathbb{R}$ denotes the reward function. $R(s, a)$ is defined only for $a \in A_s$. We consider a sparse reward setting, where the reward is only given at the end when the code emulation process is complete, and a puzzle is generated. We use the score w.r.t. $\mathcal{F}_{\text{score}}$ as the reward.
- $S_0 \subseteq S$ is the set of initial states. This can be any viable configuration of the grid. The current code trace is empty, and the code execution has not started yet.

We consider an episodic, finite horizon setting. This means that starting from an initial state s_0, the agent interacts with the environment over discrete timesteps t. The episode ends either when the code emulation finishes or the episode length exceeds a pre-specified maximum number of timesteps.

To learn the policy, we use policy gradient methods. These methods generally learn by using gradient ascent, thus updating the parameters θ of the parameterized policy $\pi_\theta(a|s)$ to increase the expected reward of the policy in the MDP. Naturally, a neural network can be used to learn the policy, where θ represents the network's weights. The network would take action a and state s as input, outputting a logit $H_\theta(a|s)$. Given the logits, we map them to a probabilistic distribution using softmax: $\pi_\theta(a|s) = \frac{\exp\left(H_\theta(a|s)\right)}{\sum_{a' \in A_s} \exp\left(H_\theta(a'|s)\right)}$. We use an Actor-Critic method for training our agent. We denote with $\hat{v}(s, w)$ the value for state s predicted by the critic with parameters w. As we operate on batches, the parameters of both the actor and the critic remain unchanged until a buffer is filled with a fixed number of episodes. Thus, for an initial state s_0 (i.e., an empty or pre-initialized task and a code, with the emulator reset), we execute the existing policy π_θ until the buffer is filled, generating several sequences of experience as tuples $(s_t, a_t, r_t)_{t=0..T}$, where T represents a variable episode length. Thus, the losses for an episode are computed as a sum over the timesteps $t \in [0, T]$ as follows, for the actor (Equation 1) and for the critic (Equation 2, employing the smooth L1 loss, denoted as $\text{L1}_{\text{smooth}}$):

$$\text{Loss}_\theta = \sum_{t=0}^{T} \left(\sum_{\tau=t}^{T} r_\tau - \hat{v}(s_t, w) \right) \cdot \nabla_\theta \log(\pi_\theta(a_t|s_t)) \tag{1}$$

$$\text{Loss}_w = \sum_{t=0}^{T} \text{L1}_{\text{smooth}} \left(\sum_{\tau=t}^{T} r_\tau,\ \hat{v}(s_t, w) \right) \tag{2}$$

Finally, θ and w are updated by using the computed losses over the entire batch, multiplied with a learning rate.

Neural architecture. We describe the architecture of the CNN-based neural model used by the puzzle generator. We employ a similar architecture for both the HoCMaze and Karel domains, as presented in Figure 12. Only the input size for the grid (i.e., D $\times$ 16 $\times$ 16, where D $=$ 12 for HoCMaze and D $=$ 14 for Karel) and code features (i.e., F, where F $=$ 9 for HoCMaze and F $=$ 12 for Karel) differ. We process the grid by 3 CNN blocks (i.e., one block composed of Conv2D, ReLU, and MaxPooling2D layers), after which we apply 5 fully connected (linear) layers, thus obtaining the grid embedding. To the grid embedding, we concatenate the code features, which are represented in a binary manner (e.g., increase in coverage, current decision type). We then pass the concatenated tensor through an additional fully-connected layer, and its output is then passed to both the action head and the value head (i.e., necessary for the Actor-Critic algorithm).

<table>
<tr><td>Input</td><td>D × 16 × 16 Grid</td><td rowspan="9">Code features, Size = F</td></tr>
<tr><td>CNN Block 1</td><td>Conv2D, kernel size = 3, padding 1, 64 × 64
ReLU
MaxPool2D, kernel size = 2, padding 0</td></tr>
<tr><td>CNN Block 2</td><td>Conv2D, kernel size = 3, padding 1, 64 × 64
ReLU
MaxPool2D, kernel size = 2, padding 0</td></tr>
<tr><td>CNN Block 3</td><td>Conv2D, kernel size = 3, padding 1, 64 × 64
ReLU
MaxPool2D, kernel size = 2, padding 0</td></tr>
<tr><td>Linear 1</td><td>CNN output size (256) × 1024</td></tr>
<tr><td>Linear 2</td><td>1024 × 512</td></tr>
<tr><td>Linear 3</td><td>512 × 256</td></tr>
<tr><td>Linear 4</td><td>256 × 128</td></tr>
<tr><td>Linear 5</td><td>128 × 32</td></tr>
<tr><td>Linear 6</td><td colspan="2">Concatenated features size (32 + F) × 8</td></tr>
<tr><td>Linear 7 (Action and Value)</td><td>8 × action space</td><td>8 × 1</td></tr>
</table>

Figure 12: Architecture of the neural model used by the puzzle generator. D denotes the depth of the input grid and F denotes the size of the code features tensor, both different for each of the HoCMaze and Karel domains.

E Experimental Evaluation with Synthetic Task Specifications

In this section, we train and evaluate NEURTASKSYN on synthetic datasets of task specifications. We consider two popular visual programming domains: *Hour of Code:Maze Challenge* by Code.org (HoCMaze) [19, 20] and Karel [21], as introduced in Sections 1 and 2. Both these programming domains have been studied extensively in the literature on program/task synthesis [39, 54, 25, 53, 38] and computing education [55, 56, 57]. We train and evaluate different variants of NEURTASKSYN to quantify the utility of individual components. We also detail the instantiations of the scoring functions, give more insight into the synthetic dataset creation process, and show the details of the training processes for both the code generator and the task generator.

E.1 Domain-specific elements

We begin by defining a few domain-specific elements that are important for evaluation in this section. As introduced in Section 2, we use two DSLs shown in Figures 13b and 13c, adapted from the DSLs in [39, 25].

E.2 Scoring Function $\mathcal{F}_{\text{score}}$

As mentioned in Section 3, we will use domain-specific scoring functions $\mathcal{F}^{\text{HoCMaze}}_{\text{score}}$ and $\mathcal{F}^{\text{Karel}}_{\text{score}}$ to capture quality of a visual programming task. In our work, we adapt scoring functions used in [25]. These scoring functions will be used in different ways throughout this section: (a) during training of NEURTASKSYN's puzzle generator as a reward for RL agent and during inference to select an output task from candidates; (b) when evaluating different techniques with a surrogate metric based on these scoring functions; (c) when creating synthetic dataset.

Next, we describe the two instantiations for $\mathcal{F}_{\text{score}}$ as used in the two domains HoCMaze and Karel. We adopt a scoring function similar to that of [25], where $\mathcal{F}_{\text{score}}$ is used for guiding a Monte Carlo Tree Search, as an evaluation function that describes the desired properties of their system's output. We note that our method can work with any other instantiation of the scoring function $\mathcal{F}_{\text{score}}$. The instantiations we use for $\mathcal{F}_{\text{score}}$ for each of the HoCMaze and Karel domains are defined in Equations 3 and 4 and are comprised of different components: (i) $\mathcal{F}_{\text{cov}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) \in [0, 1]$ computes the coverage ratio, i.e., ratio of executed blocks to total number of blocks; (ii) $\mathcal{F}_{\text{sol}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) \in \{0, 1\}$ evaluates to 1 if $\texttt{C}^{\text{out}}$ correctly solves $\texttt{T}^{\text{out}}_{\text{puzzle}}$, i.e., no crashing, reaching the goal/converting the pre-grid to the post-grid; (iii) $\mathcal{F}_{\text{nocross}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) \in [0, 1]$ computes the ratio of cells visited exactly once with regard to the total number of visited cells; (iv) $\mathcal{F}_{\text{nocut}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) \in \{0, 1\}$ evaluates to 0 if there is a shortcut sequence comprised of basic actions; (v) $\mathcal{F}_{\text{notred}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) \in \{0, 1\}$ evaluates to 0 if there are redundant action sequences in $\texttt{C}^{\text{out}}$, e.g., sequences like `turnLeft`, `turnRight`, or if the codes obtained by eliminating one action, loop or conditional from $\texttt{C}^{\text{out}}$ solves $\texttt{T}^{\text{out}}_{\text{puzzle}}$; (vi) $\mathcal{F}_{\text{qual}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) \in [0, 1]$ evaluates the visual quality of $\texttt{T}^{\text{out}}_{\text{puzzle}}$ as per Equation 5; (vii) $\mathcal{F}_{\text{cutqual}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) \in [0, 1]$ evaluates visual quality of the shortest path made only of basic actions, similar to $\mathcal{F}_{\text{qual}}$. We set $\alpha_1 = \alpha_2 = \frac{1}{2}$ and $\alpha_3 = \alpha_4 = \alpha_5 = \frac{1}{3}$.

$$\begin{aligned}\mathcal{F}^{\text{HoCMaze}}_{\text{score}}(\texttt{T}^{\text{out}}, \texttt{C}^{\text{out}}) =& \mathbb{1}\Big[\mathcal{F}_{\text{cov}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) = 1, \mathcal{F}_{\text{sol}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) = 1, \mathcal{F}_{\text{nocross}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) = 1,\\ &\mathcal{F}_{\text{nocut}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) = 1, \mathcal{F}_{\text{notred}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) = 1\Big]\cdot\\ &\Big[\alpha_1\mathcal{F}_{\text{cov}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) + \alpha_2\mathcal{F}_{\text{qual}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}})\Big]\end{aligned} \tag{3}$$

$$\begin{aligned}\mathcal{F}^{\text{Karel}}_{\text{score}}(\texttt{T}^{\text{out}}, \texttt{C}^{\text{out}}) =& \mathbb{1}\Big[\mathcal{F}_{\text{cov}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) = 1, \mathcal{F}_{\text{sol}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) = 1, \mathcal{F}_{\text{nocross}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) = 1,\\ &\mathcal{F}_{\text{nocut}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) = 1, \mathcal{F}_{\text{notred}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) = 1\Big]\cdot\\ &\Big[\alpha_3\mathcal{F}_{\text{cov}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) + \alpha_4\mathcal{F}_{\text{qual}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}}) + \alpha_5\mathcal{F}_{\text{cutqual}}(\texttt{T}^{\text{out}}_{\text{puzzle}}, \texttt{C}^{\text{out}})\Big]\end{aligned} \tag{4}$$

Domain	All	Easy (depth, constructs)			Hard (depth, constructs)	
		$(1,0)$	$(2,1)$	$(2,2)$	$(3,2)$	$(3,3)$
HoCMaze	$1,016$	183	69	47	136	581
Karel	$1,027$	300	155	277	295	0

(a) Dataset of synthetic task specifications

```
code C     := def RUN () DO y
rule y     := s | g | s; g
rule s     := a | s; s | IF (b) DO s | IF (b) DO s ELSE s
              | REPEAT (x) DO s
rule g     := REPEATUNTIL (goal) DO s
action a   := move | turnLeft | turnRright
bool b     := pathAhead | pathLeft | pathRight
iter x     := 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
```

(b) DSL for HoCMaze domain

```
code C     := def RUN () DO s
rule s     := a | s; s | IF (b) DO s | IF (b) DO s ELSE s
              | WHILE (b) DO s | REPEAT (x) DO s
action a   := move | turnLeft | turnRright | putMarker | pickMarker
bool b     := pathAhead | pathLeft | pathRight
              | no-pathAhead | markerPresent | no-markerPresent
iter x     := 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
```

(c) DSL for Karel domain

Figure 13: **(a)** Synthetic datasets used for training/evaluation. **(b)** DSLs for two domains.

We use the same measure of visual quality for both domains, keeping into account the number of moves, turns, segments, long-segments, and turn-segments, as explained next. More specifically, segments and long-segments correspond to consecutive sequences of 'moves' containing more than 3 and 5 actions, respectively; turn-segments correspond to consecutive sequences of 'turnLeft' or 'turnRight' containing more than 3 actions. The formula for $\mathcal{F}_{\text{qual}}$ is given in Equation 5 below; we also clip the values for each counter # w.r.t. its corresponding normalization factor (not depicted here for brevity).

$$\begin{aligned}\mathcal{F}_{\text{qual}}(\texttt{T}^{\text{out}}, \texttt{C}^{\text{out}}) = & \frac{3}{4} \cdot \left(\frac{1}{4} \cdot \left(\frac{\#\text{moves}}{2n} + \frac{\#\text{turns}}{n} + \frac{\#\text{segments}}{n/2} + \frac{\#\text{long-segments}}{n/3}\right)\right) + \\ & \frac{1}{4} \cdot \left(1 - \frac{\#\text{turn-segments}}{n/2}\right)\end{aligned} \tag{5}$$

E.3 Synthetic task specifications

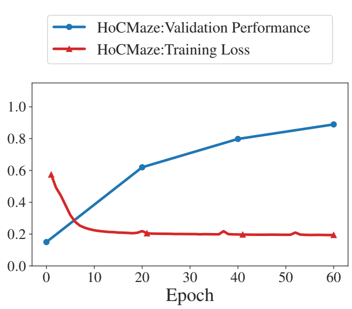
For training and evaluation of techniques, we create a dataset of synthetic task specifications per domain, referred to as $\mathbb{D} := \{\psi^{\text{in}}\}$. Figure 13a provides a summary of datasets $\mathbb{D}$ for each domain. Next, we provide an overview of how we create a dataset per domain. To create one specification $\psi^{\text{in}} := (\psi^{\text{in}}_{\text{puzzle}}, \psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta}, \psi^{\text{in}}_{\text{size}})$, the most crucial part is getting a code sketch $\psi^{\text{in}}_{\text{sketch}}$ that respects the DSL and can lead to a valid code generation. We start by sampling a code $\texttt{C}^{\text{in}}$ from the DSL for a given structure, depth, and constructs – this sampling process is inspired by methods for synthetic dataset creation [39, 54, 25]. We follow Algorithm 1 to create dataset $\mathbb{D}$. For each sampled code, we check its semantic validity, i.e., this code can lead to a high-quality task. For this purpose, we make use of an offline, time-intensive, method TASKORACLE($\texttt{C}$): it does one million symbolic executions of a given code $\texttt{C}$ and returns highest-scoring task w.r.t. scoring function $\mathcal{F}_{\text{score}}$. We filter them out if a low-quality task is obtained, supplementing this filtering with an additional inspection step. This inspection step is necessary because semantic irregularities (e.g., `IfElse` with the same `If` and `Else` bodies) can get past the previous filtering step. As the compute and implementation efforts are larger for an automatic system that would detect such irregularities, which are easy to spot, we opt for a direct inspection step. Afterwards, for a sampled code $\texttt{C}^{\text{in}}$ for which TASKORACLE($\texttt{C}^{\text{in}}$) succeeds, we create its corresponding $\psi^{\text{in}}_{\text{sketch}}$ by keeping only the programming constructs (loops/conditions) with a random subset of the booleans/iterators masked out. The rest of the ψ^{in} elements are instantiated as follows: $\psi^{\text{in}}_{\text{puzzle}}$ is 16×16 size without any initialization, $\psi^{\text{in}}_{\Delta}$ only allows filling in the sketch with basic actions (without any constructs) and booleans/iterators, and $\psi^{\text{in}}_{\text{size}}$ is randomly initialized in the range $[\texttt{C}^{\text{in}}_{\text{nblock}}, 17]$. In our evaluation, we split $\mathbb{D}$ as follows: 80% for training the neural models ($\mathbb{D}^{\text{train}}$), 10% for validation ($\mathbb{D}^{\text{val}}$), and a fixed 10% for evaluation ($\mathbb{D}^{\text{test}}$).

Algorithm 1: Specification Dataset Collection Procedure

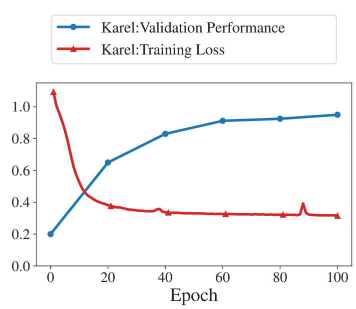
Input: list $\mathbb{S}$ of tuples (code structure s, required size l); maximum candidate set size m;
$\mathbb{D} \leftarrow \emptyset$; `/* Dataset initialized to empty set */`
foreach $(s, l) \in \mathbb{S}$ **do**
  $\mathbb{C} \leftarrow \emptyset$; `/* Candidate set initialized to empty set */`
  while size($\mathbb{C}$) $< m$ **do**
    code $\leftarrow$ GenerateCode(s);
    task $\leftarrow$ TASKORACLE(code);
    score $\leftarrow$ $\mathcal{F}_{\text{score}}$(task, code);
    if score > 0 **then**
      add (code, task, score) to $\mathbb{C}$;
  sort $\mathbb{C}$ according to the score, in decreasing order;
  counter $\leftarrow$ 0;
  while counter $< l$ and $\mathbb{C} \neq \emptyset$ **do**
    (code, task, score) $\leftarrow$ Pop($\mathbb{C}$);
    accept $\leftarrow$ Inspect(task, code) ; `/* Inspection step */`
    if accept **then**
      $\psi \leftarrow$ ExtractSpecs(code);
      add ψ to $\mathbb{D}$;
      counter $\leftarrow$ counter+1;
Output: Dataset $\mathbb{D}$;

E.4 Training Process

Training the code generator. We employ a standard approach, using an imitation (supervised) form of learning, with a cross-entropy loss for an LSTM-based architecture (see Appendix D) [49, 39]. We augment $\mathbb{D}^{\text{train}}$ by adding all the possible combinations of construct instantiations for a given code. The training plots and the hyperparameters used can be seen in Figure 14. We report the validation performance (i.e., same metric employed for NEURCODEGEN) smoothed via an exponential decay function, and the batch loss averaged over one epoch.



(a) HoCMaze



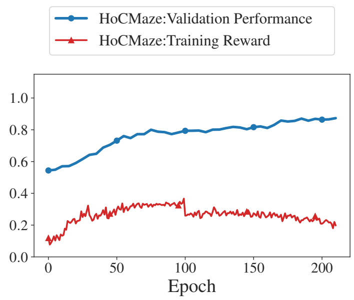
(b) Karel

HyperParameter	HoCMaze	Karel
Max. Epochs	60	100
Batch Size	32	32
Learning Rate	5×10^{-4}	5×10^{-4}
Dict. Size	59	59
Max. Blocks	17	17

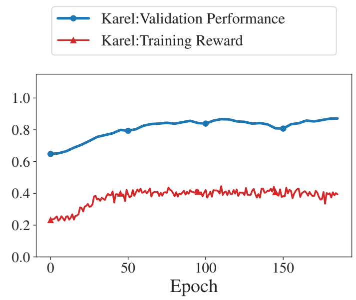
(c) Hyperparameters

Figure 14: Illustration of training details for the code generator. (a) and (b) show the training curves with mean epoch loss and validation performance, based on metric $\mathcal{M}$, for both the HoCMaze and Karel domains. (c) shows the hyperparameters employed for the code generator training.

Training the puzzle generator. We use an RL procedure, using the instantiations of $\mathcal{F}_{\text{score}}$ as rewards. We augment the RL training set with additional codes produced by the previously trained code generator. To encourage higher quality tasks, we use a form of curriculum as follows: after a certain epoch, we give a reward larger than 0 only if the ratio between the scores of the output task and the TASKORACLE's task is larger than a factor $\widehat{\lambda}_2$; we gradually increase $\widehat{\lambda}_2$ from 0.8 to 0.9. For Karel, we also employ a temperature parameter during training, encouraging exploration during inference. The training plots and the hyperparameters used can be seen in Figures 15. We report the validation performance (i.e., same metric employed for NEURPUZZLEGEN) smoothed via an exponential decay function, and the batch reward averaged over one epoch.



(a) HoCMaze



(b) Karel

HyperParameter	HoCMaze	Karel
Max. Epochs	500	500
Batch Size	32	32
Learning Rate	10^{-4}	10^{-4}
Temperature	1	0.5
Curriculum	True	True

(c) Hyperparameters

Figure 15: Illustration of training details for the puzzle generator. (a) and (b) show the training curves with mean epoch reward and validation performance, based on metric $\mathcal{M}$ for both the HoCMaze and Karel domains. A form of curriculum learning was employed, which explains the lack of general monotonicity for the reward. (c) shows the hyperparameters employed for the puzzle generator training.

Further implementation details. We limit the number of possible initial locations for a grid to one representative per quadrant. In total, we consider 5 quadrants (i.e., top-left, bottom-left, center, top-right, bottom-right). We do this to limit the action space to a more tractable amount for a variable grid size. With 5 quadrants and 4 possible orientations, this leads to $5 \times 4 = 20$ possible initial location/orientation pairs, offering already enough variability.

F Evaluation on the synthethic dataset

Techniques evaluated. First we describe NeurTaskSyn, our main task synthesis technique from Section 3. For each domain (HoCMaze and Karel), we train a separate instance of NeurTaskSyn using the synthetic dataset introduced above. In Section 3, we described the generation process for a single "rollout", i.e., one $\texttt{C}^{\text{out}}$ and one puzzle $\texttt{T}^{\text{out}}_{\text{puzzle}}$ is generated. In practice, we use multiple rollouts to select a final output task $\texttt{T}^{\text{out}}$. More concretely, at inference time for a given ψ^{in} as input, NeurTaskSyn generation process is captured by two parameters: number of code rollouts c by the code generator and number of puzzle rollouts p by the puzzle generator for each generated code. We denote these hyperparameters in subscript, e.g., NeurTaskSyn$_{\text{c:5,p:10}}$ for 5×10 rollouts. Out of these $\text{c} \times \text{p}$ candidates, the technique outputs one task $\texttt{T}^{\text{out}}$ along with solution code $\texttt{C}^{\text{out}}$ using its scoring function. Next, we describe different variants of NeurTaskSyn$_{\text{c,p}}$ and baselines:

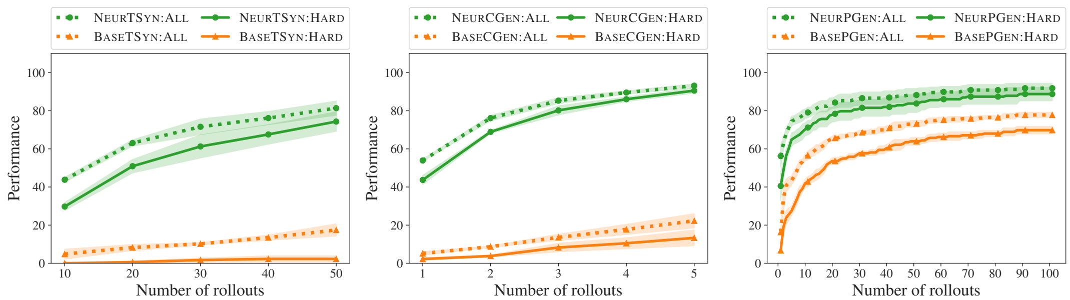
- NeurCodeGen$_{\text{c,p:OPT}}$: This technique is a variant of NeurTaskSyn$_{\text{c,p}}$ to evaluate its code generation component, assuming access to high quality puzzle generator. More concretely, we replace the puzzle generator component of NeurTaskSyn with TaskOracle used for creating synthetic dataset. At inference time, NeurCodeGen$_{\text{c,p:OPT}}$ generation process is captured by hyperparameter c, i.e., the number of code rollouts; we use TaskOracle to generate a puzzle for each generated code. Out of c candidates, the technique outputs one task analogous to NeurTaskSyn.
- NeurPuzzleGen$_{\text{c:FIX,p}}$: This technique is a variant of NeurTaskSyn$_{\text{c,p}}$ to evaluate its puzzle generation component, assuming the code generator has access to code $\texttt{C}^{\text{in}}$ associated with specification ψ^{in} in the dataset. At inference time, NeurPuzzleGen$_{\text{c:FIX,p}}$ generation process is captured by hyperparameter p, i.e., the number of puzzle rollouts. Out of p candidates, the technique outputs one task analogous to NeurTaskSyn.
- BaseTaskSyn$_{\text{c,p}}$, BaseCodeGen$_{\text{c,p:OPT}}$, BasePuzzleGen$_{\text{c:FIX,p}}$: These techniques operate similar to NeurTaskSyn$_{\text{c,p}}$ and its variants, but use only symbolic engine with random search.[2]

Evaluation metrics. Next we introduce a binary success metric $\mathcal{M}(\psi^{\text{in}}, \texttt{T}^{\text{out}}, \texttt{C}^{\text{out}})$ that is used to compare the performance of different techniques and to pick hyperparameters. More concretely, $\mathcal{M}(\psi^{\text{in}}, \texttt{T}^{\text{out}}, \texttt{C}^{\text{out}})$ is 1 if the following hold: (i) the task $\texttt{T}^{\text{out}}$ is valid w.r.t. ψ^{in} as per objective O1:Validity in Section 2; (ii) the generated code $\texttt{C}^{\text{out}}$ is semantically correct in a sense that it

[2] TaskOracle($\texttt{C}$) introduced above uses BasePuzzleGen with $\text{p} = 10^6$ rollouts for a fixed code $\texttt{C}$.

Technique	HoCMaze			Karel		
	All	Easy	Hard	All	Easy	Hard
BaseTaskSyn$_{c:5,p:10}$	13.6 (1.3)	46.2 (4.1)	2.0 (1.3)	35.7 (1.0)	49.3 (0.9)	10.9 (2.7)
NeurTaskSyn$_{c:5,p:10}$	81.4 (3.7)	100.0 (0.0)	74.3 (5.1)	92.6 (1.4)	100.0 (0.0)	79.3 (3.9)
BaseCodeGen$_{c:5,p:OPT}$	22.3 (3.8)	47.8 (5.4)	13.2 (4.0)	40.8 (2.4)	51.7 (0.5)	20.8 (6.2)
NeurCodeGen$_{c:5,p:OPT}$	93.1 (1.0)	100.0 (0.0)	90.5 (1.4)	98.1 (0.6)	100.0 (0.0)	94.6 (1.6)
BasePuzzleGen$_{c:FIX,p:10}$	55.6 (1.8)	91.7 (2.4)	41.9 (2.3)	71.8 (3.8)	86.6 (3.8)	45.0 (3.9)
NeurPuzzleGen$_{c:FIX,p:10}$	78.4 (2.5)	100.0 (0.0)	70.3 (3.4)	79.8 (0.6)	92.0 (1.3)	57.7 (1.8)

Figure 16: Results on synthetic task specifications for HoCMaze and Karel; see Figure 13a.



(a) BaseTaskSyn, NeurTaskSyn (b) BaseCodeGen, NeurCodeGen (c) BasePuzzleGen, NeurPuzzleGen

Figure 17: **(a)** Results for BaseTaskSyn$_{c,p}$ and NeurTaskSyn$_{c,p}$ by increasing code rollouts c from 1 to 5 with fixed puzzle rollouts $p = 10$. **(b)** Results for BaseCodeGen$_{c,p:OPT}$ and NeurCodeGen$_{c,p:OPT}$ by increasing code rollouts c from 1 to 5. **(c)** Results for BasePuzzleGen$_{FIX,p}$ and NeurPuzzleGen$_{FIX,p}$ by increasing puzzle rollouts p from 1 to 100.

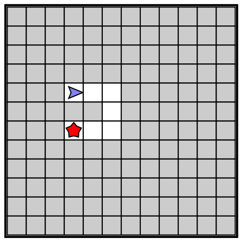
can lead to a valid task via TaskOracle, i.e., $\mathcal{F}_{\text{score}}(\text{TaskOracle}(\texttt{C}^{\text{out}}), \texttt{C}^{\text{out}}) > \lambda_1$; (iii) the generated task $\texttt{T}^{\text{out}}$ is good quality in comparison to the oracle-generated task, i.e., $\mathcal{F}_{\text{score}}(\texttt{T}^{\text{out}}, \texttt{C}^{\text{out}}) > \lambda_2 \cdot \mathcal{F}_{\text{score}}(\text{TaskOracle}(\texttt{C}^{\text{out}}), \texttt{C}^{\text{out}})$. We use $\lambda_1 = 0$ and $\lambda_2 = 0.9$ in our experiments. For each technique, performance is computed as % success rate across $\mathbb{D}^{\text{test}}$ w.r.t. $\mathcal{M}$; in total, we compute performance across three seeds and report averaged results as mean (stderr). Importantly, we note that this metric only serves as a surrogate metric for evaluation on synthetic dataset; the neural models trained here will be evaluated on real-world task specifications w.r.t. the sythesis objectives in the next section.

Results. Figure 16 reports evaluation results for different techniques for a fixed number of code/puzzle rollouts across two domains and segments. Figure 17 further report results as we vary the rollouts for different techniques. In summary, these results demonstrate the utility of different components of NeurTaskSyn and how the synthesis quality improves as we increase the number of rollouts.

G Experimental Evaluation with Real-World Task Specifications: Details

G.1 Real-World Task Specifications

In Figures 18 and 19 below, we list the source tasks `T` and codes `C` for the 10 task specifications mentioned in Figure 3.



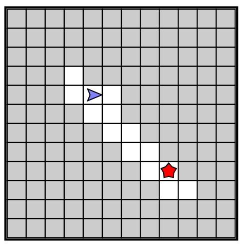
T_{store} = {`move`, `turnLeft`, `turnRight`, REPEAT}

$T_{size} = 5$

(a) Source `T` for ψ_0

```
def RUN(){
  REPEAT(3){
    move
    move
    turnRight
  }
}
```

(b) Source `C` for ψ_0



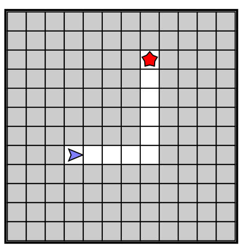
T_{store} = {`move`, `turnLeft`, `turnRight`, REPEATUNTIL}

$T_{size} = 6$

(c) Source `T` for ψ_1

```
def RUN(){
  REPEATUNTIL(goal){
    turnRight
    move
    turnLeft
    move
  }
}
```

(d) Source `C` for ψ_1



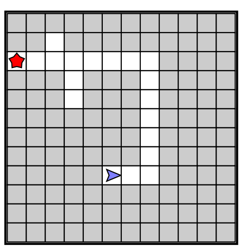
T_{store} = {`move`, `turnLeft`, `turnRight`, REPEAT}

$T_{size} = 6$

(e) Source `T` for ψ_2

```
def RUN(){
  REPEAT(4){
    move
  }
  turnLeft
  REPEAT(5){
    move
  }
}
```

(f) Source `C` for ψ_2



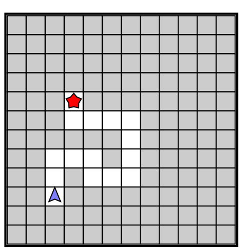
T_{store} = {`move`, `turnLeft`, `turnRight`, REPEATUNTIL, IFELSE}

$T_{size} = 5$

(g) Source `T` for ψ_3

```
def RUN(){
  REPEATUNTIL(goal){
    IF(pathAhead){
      move
    }
    ELSE{
      turnLeft
    }
  }
}
```

(h) Source `C` for ψ_3



T_{store} = {`move`, `turnLeft`, `turnRight`, REPEATUNTIL, IF}

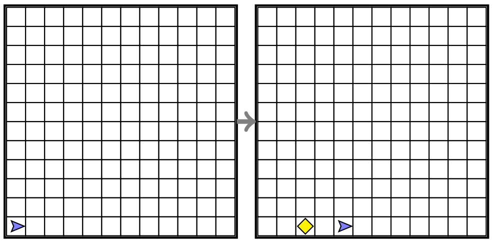
$T_{size} = 7$

(i) Source `T` for ψ_4

```
def RUN(){
  REPEATUNTIL(goal){
    move
    IF(pathLeft){
      turnLeft
    }
    IF(pathRight){
      turnRight
    }
  }
}
```

(j) Source `C` for ψ_4

Figure 18: Overview of HoCMaze sources.



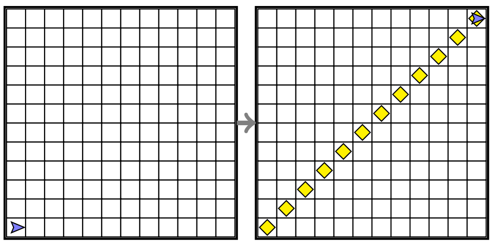
T_{store} = {move, turnLeft, turnRight, pickMarker, putMarker}

T_{size} = 6

(a) Source T for ψ_5

```
def RUN(){
  move
  move
  putMarker
  move
  move
}
```

(b) Source C for ψ_5



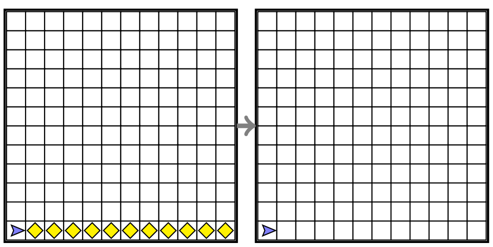
T_{store} = {move, turnLeft, turnRight, pickMarker, putMarker, WHILE}

T_{size} = 8

(c) Source T for ψ_6

```
def RUN(){
  putMarker
  WHILE(pathAhead){
    move
    turnLeft
    move
    turnRight
    putMarker
  }
}
```

(d) Source C for ψ_6



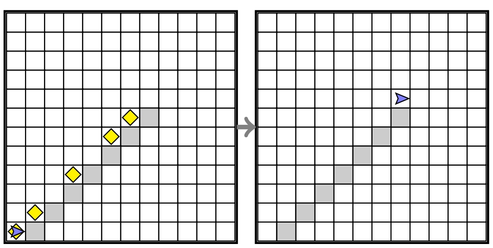
T_{store} = {move, turnLeft, turnRight, pickMarker, putMarker, WHILE}

T_{size} = 10

(e) Source T for ψ_7

```
def RUN(){
  WHILE(pathAhead){
    move
    pickMarker
  }
  turnLeft
  turnLeft
  WHILE(pathAhead){
    move
  }
  turnLeft
  turnLeft
}
```

(f) Source C for ψ_7



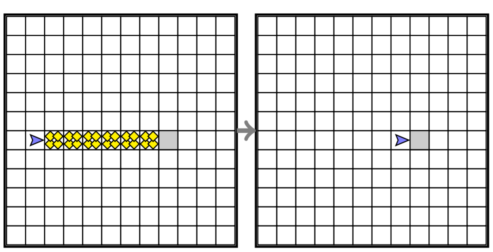
T_{store} = {move, turnLeft, turnRight, pickMarker, putMarker, WHILE, IF}

T_{size} = 8

(g) Source T for ψ_8

```
def RUN(){
  WHILE(no-pathAhead){
    IF(markerPresent){
      pickMarker
    }
    turnLeft
    move
    turnRight
    move
  }
}
```

(h) Source C by ψ_8



T_{store} = {move, turnLeft, turnRight, pickMarker, putMarker, WHILE, REPEAT}

T_{size} = 5

(i) Source T for ψ_9

```
def RUN(){
  WHILE(pathAhead){
    move
    REPEAT(4){
      pickMarker
    }
  }
}
```

(j) Source C by ψ_9

Figure 19: Overview of Karel sources.

G.2 GPT4TaskSyn

We describe next the details of our interaction with GPT-4 for generating the visual puzzles $\texttt{T}^{\text{out}}_{\text{puzzle}}$ and the intermediate code $\texttt{C}^{\text{out}}$ via the GPT4TaskSyn technique. Our interaction is conducted through the platform [12]. We try several strategies and prompts to make GPT-4 work for synthesizing visual programming tasks, as it tends to struggle with logical and spatial reasoning. Thus, we opt for a two-stage task synthesis process which works the best. We first ask GPT-4 to generate a code $\texttt{C}^{\text{out}}$ for ψ^{in}, by using 5 separate queries. We start with an initial prompt and then use follow-up prompts to fix any mistakes, as GPT-4 occasionally ignores part of the specifications. The initial and follow-up prompts used for generating $\texttt{C}^{\text{out}}$ are presented in Figures 20a and 21a. We select the best code generated during the 5 separate queries based on our expertise. The second stage comprises of additional 5 separate queries for generating $\texttt{T}^{\text{out}}_{\text{puzzle}}$ for the selected code $\texttt{C}^{\text{out}}$. Again, we start with an initial prompt and then use follow-up prompts to fix any issues. The follow-up prompts are necessary because GPT-4 tends to struggle with spatial orientation and with the relationship between $\texttt{C}^{\text{out}}$ and $\texttt{T}^{\text{out}}_{\text{puzzle}}$. The initial and follow-up prompts used for generating $\texttt{T}^{\text{out}}_{\text{puzzle}}$ are presented in Figures 20b and 21b. Similar to the code selection process, we select the best visual puzzle generated during the 5 separate queries based on our expertise. Once we get $\texttt{C}^{\text{out}}$ and $\texttt{T}^{\text{out}}_{\text{puzzle}}$, we set other elements of the task, $\texttt{T}^{\text{out}}_{\text{store}}$ and $\texttt{T}^{\text{out}}_{\text{size}}$, as done for BaseTaskSyn and NeurTaskSyn.

Code: Initial prompt

I am working in the block-based visual programming domain of Hour of Code: Maze Challenge from code.org. In this domain, the following types of coding blocks are available:

- Basic action blocks: move forward, turn left, turn right.
- Boolean conditions: path ahead, path left, path right.
- Loops: repeatUntil(goal), repeat(int).
- Conditionals: if(boolean), if(boolean)else.

In this domain, a task is represented as an 8x8 visual grid that contains WALL cells, FREE cells, AVATAR (with specific location and direction), and GOAL. We represent a task's 8x8 visual grid with the following symbols.

represents a WALL cell.

+ represents a FREE cell.

* represents GOAL.

E represents AVATAR's location facing East direction.

W represents AVATAR's location facing West direction.

N represents AVATAR's location facing North direction.

S represents AVATAR's location facing South direction.

Below, I am giving you a program structure. Can you generate a code that respects this program structure?

— Structure —

[SKETCH]

You should not change the structure. This means that you shouldn't add or remove any loops (e.g., repeatUntil(goal), repeat(int)) and conditionals (e.g., if(boolean), if(boolean)else). The program needs to be valid, meaning that bodies of constructs cannot remain empty. To complete this given structure, you can use basic action blocks, boolean conditions, and iteration numbers that are available in the Hour of Code: Maze Challenge programming.

— Code —

Code: Follow-up prompt in case of constructs changed

Your code does not follow the program structure I have given. You shouldn't add or remove any loops (e.g., repeatUntil(goal), repeat(int)) and conditionals (e.g., if(boolean), if(boolean)else). Can you try to generate a new code for the same structure?

Code: Follow-up prompt for any other issues

Your code could be improved! You can think of producing a better code by reasoning about the AVATAR's actions when the code is executed. Can you try to generate a new code respecting the program structure I have given?

(a) Prompts used for obtaining $\texttt{C}^{\text{out}}$

Task: Initial prompt

I am working in the block-based visual programming domain of Hour of Code: Maze Challenge from code.org. In this domain, the following types of coding blocks are available:

- Basic action blocks: move forward, turn left, turn right.

- Boolean conditions: path ahead, path left, path right.

- Loops: repeatUntil(goal), repeat(int).

- Conditionals: if(boolean), if(boolean)else.

In this domain, a task is represented as an 8x8 visual grid that contains WALL cells, FREE cells, AVATAR (with specific location and direction), and GOAL. We represent a task's 8x8 visual grid with the following symbols.

represents a WALL cell.

+ represents a FREE cell.

* represents GOAL.

E represents AVATAR's location facing East direction.

W represents AVATAR's location facing West direction.

N represents AVATAR's location facing North direction.

S represents AVATAR's location facing South direction.

Below I am giving you a solution code. Can you generate a task with 8x8 visual grid that would be solved by this code?

— Solution —

[CODE]

The visual grid must contain AVATAR (with specific location and direction) along with GOAL, and can have WALL cells and FREE cells. Number your grid with row numbers (1 to 8) and column numbers (1 to 8). Also, you should tell me the position of AVATAR and GOAL in your generated task so we are sure about the numbering.

You can verify the correctness of your generated task by executing the solution code on your task. A solution code for a task takes AVATAR to GOAL when executed. Note that AVATAR can only move on FREE cells and will crash if it tries to go to a WALL cell. If your generated task is not correct, you should try again to generate a correct task.

— Task —

Task: Follow-up prompt for any issues

Your code does not solve the generated grid. Be careful with the AVATAR as it should reach the goal after the code execution. Keep the code fixed. Can you try to generate a new visual grid and explain your reasoning? Recall that your code, when executed, should take the AVATAR from its initial location to the GOAL.

(b) Prompts used for obtaining the main part of $\mathrm{T}^{\text{out}}_{\text{puzzle}}$

Figure 20: Prompts used in the implementation of GPT4TaskSyn technique for HoCMaze domain.

Code: Initial prompt

I am working in the block-based visual programming domain of Karel programming. In this domain, the following types of coding blocks are available:

- Basic action blocks: move forward, turn left, turn right, pick marker, put marker.

- Boolean conditions: path ahead, path left, path right, marker present, no path ahead, no marker present.

- Loops: while(boolean), repeat(int).

- Conditionals: if(boolean), if(boolean)else.

In this domain, a task is represented as a pair of 10x10 visual pregrid and 10x10 visual postgrid. This pregrid and postgrid contain WALL cells, FREE cells, AVATAR (with specific location and direction), and markers. We represent a task's 10x10 visual pregrid and postgrid with the following symbols.

represents a WALL cell.

+ represents a FREE cell.

m represents a cell with marker.

E represents AVATAR's location on a cell without marker, facing East direction.

W represents AVATAR's location on a cell without marker, facing West direction.

N represents AVATAR's location on a cell without marker, facing North direction.

S represents AVATAR's location on a cell without marker, facing South direction.

Em represents AVATAR's location on a cell with marker, facing East direction.

Wm represents AVATAR's location on a cell with marker, facing West direction.

Nm represents AVATAR's location on a cell with marker, facing North direction.

Sm represents AVATAR's location on a cell with marker, facing South direction.

Below, I am giving you a program structure. Can you generate a code that respects this program structure?

—- Structure —-

[SKETCH]

You should not change the structure. This means that you shouldn't add or remove any loops (e.g., while(boolean), repeat(int)) and conditionals (e.g., if(boolean), if(boolean)else). The program needs to be valid, meaning that bodies of constructs cannot remain empty. To complete this given structure, you can use basic action blocks, boolean conditions, and iteration numbers that are available in Karel programming.

— Code —

Code: Follow-up prompt in case of constructs changed

Your code does not follow the programming structure I have given. You shouldn't add or remove any loops (e.g., while(boolean), repeat(int)) and conditionals (e.g., if(boolean), if(boolean)else). Can you try to generate a new code for the same structure?

Code: Follow-up prompt for any other issues

Your code could be improved! You can think of producing a better code by reasoning about the Karel AVATAR when the code is executed. Can you try to generate a new code?

(a) Prompts used for obtaining $\texttt{C}^{\text{out}}$

Task: Initial prompt

I am working in the block-based visual programming domain of Karel programming. In this domain, the following types of coding blocks are available:

- Basic action blocks: move forward, turn left, turn right, pick marker, put marker.
- Boolean conditions: path ahead, path left, path right, marker present, no path ahead, no marker present.
- Loops: while(boolean), repeat(int).
- Conditionals: if(boolean), if(boolean)else.

In this domain, a task is represented as a pair of 10x10 visual pregrid and 10x10 visual postgrid. This pregrid and postgrid contain WALL cells, FREE cells, AVATAR (with specific location and direction), and markers. We represent a task's 10x10 visual pregrid and postgrid with the following symbols.

represents a WALL cell.

+ represents a FREE cell.

m represents a cell with marker.

E represents AVATAR's location on a cell without marker, facing East direction.

W represents AVATAR's location on a cell without marker, facing West direction.

N represents AVATAR's location on a cell without marker, facing North direction.

S represents AVATAR's location on a cell without marker, facing South direction.

Em represents AVATAR's location on a cell with marker, facing East direction.

Wm represents AVATAR's location on a cell with marker, facing West direction.

Nm represents AVATAR's location on a cell with marker, facing North direction.

Sm represents AVATAR's location on a cell with marker, facing South direction.

Below I am giving you a solution code. Can you generate a task with a pair of 10x10 visual pregrid and 10x10 visual postgrid that would be solved by this code?

— Solution —

[CODE]

Both the visual pregrid and visual postgrid must contain AVATAR (with specific location and direction), and can have WALL cells, FREE cells, and markers. Number your grids with row numbers (1 to 10) and column numbers (1 to 10). Also, you should tell me the position of AVATAR in your generated pregrid and postgrid so we are sure about the numbering.

You can verify the correctness of your generated task by executing the solution code on your task. A solution code for a task transforms the pregrid into the postgrid when executed. Note that AVATAR can only move on FREE cells and will crash if it tries to go to a WALL cell. If your generated task is not correct, you should try again to generate a correct task.

— Task —

Task: Follow-up prompt for any issues

Your code does not solve the generated pregrid and postgrid. Be careful with the AVATAR in the postgrid as it should show the effect of the code execution. Keep the code fixed. Can you try to generate a new visual pregrid and postgrid and explain your reasoning? Recall that your code, when executed, should transform the pregrid into the postgrid. Be careful with the AVATAR in the postgrid as it should show the effect of the code execution.

(b) Prompts used for obtaining the main part of $\mathrm{T}^{\text{out}}_{\text{puzzle}}$

Figure 21: Prompts used in the implementation of GPT4TaskSyn technique for Karel domain.