SYNTHESIZING OBJECT MODELS FROM NATURAL LANGUAGE SPECIFICATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Program synthesis has traditionally excelled in tasks with precise specifications such as input-output examples and formal constraints by using structured and algorithmic approaches based on enumerative search and type inference. However, traditional synthesis techniques have no mechanism of incorporating real-world knowledge, which is commonplace in software engineering. Motivated by this, we introduce a new synthesis task known as specification reification: synthesizing concrete realizations of vague, high-level application specifications. We focus on a specific instance of this: generating object models from natural language application descriptions. Towards this goal, we present three approaches for object model synthesis that leverage domain knowledge from the GPT-3 language model. In addition, we design a scoring metric to evaluate the success of synthesized object models on seven sample tasks such as classroom management and pet store applications. We demonstrate that our language-model-based synthesizers generate object models that are comparable in quality to human-generated ones.

1 INTRODUCTION

The goal of program synthesis is to generate programs from various types of specifications such as input-output examples, abstract properties about inputs and outputs, and formal logical specifications. Traditionally, program synthesis has been principally algorithmic, using methods like enumerative search, constraint-based sketching, or type-driven synthesis to generate programs. While these algorithms have been successful at synthesizing functions with well defined input/output behavior, creating software requires much more than implementing functions from well defined specifications. In particular, an important part of software development is leveraging domain knowledge to turn high-level application requirements into a detailed description of all the components and interfaces that will make up the application.

In this paper, we introduce *specification reification* as a new challenge for program synthesis. Specification reification refers to the previously mentioned process of taking a high-level, potentially vague specification of a problem and reifying it into a more concrete form. For example, consider a developer who is designing a classroom management application in an object-oriented language. Traditional program synthesis could help implement specific functions in this application—for example, a function to search for students who have not submitted an assignment—but before a developer gets to that point, they first need to design the application itself. This involves deciding which objects they need, and for each object deciding on their fields and methods, and for each method deciding what its specification should be. Specification reification is the process of deriving this design from the high-level description of the application.

We consider a specific sub-problem of specification reification: the task of synthesizing object models from natural language specifications. More concretely, we present a system that takes as input a description of an application—e.g. *"I want a classroom management app that tracks students, the assignments they've submitted, and the grades they've earned on those assignments"*—and from this description synthesizes an object model consisting of a set of objects and their fields. A sample object model that one may wish to synthesize is shown in Figure 1.

We emphasize two challenges of the task at hand:

student	submission	assignment	teacher
- name - email address - teacher - grade - assignments - submissions	- student - assignment - grade	- title - description - due date	- name - students - submissions

Figure 1: A sample object model for a classroom management application

1. First, it is challenging to automatically inferring objects and fields. In Figure 1, the *teacher* and *submission* objects are not given in the prompt; nor are the *name* and *submissions* fields for the *student table*—both need to be inferred. A successful system for this task should be able to infer related objects directly from the prompt.
2. Second, many objects refer to other objects in their fields: for example, the *student* object references the *teacher* object, the *assignment* object, and the *submission* object as fields. This highlights that a system should understand how various objects are related.

Towards this goal, our contributions are as follows:

1. We introduce and highlight a new task in program synthesis: specification reification. We also introduce the object model synthesis task as an important sub-problem of specification reification.
2. We introduce a new evaluation metric and benchmark for object model synthesis. First, we gather a small dataset of hand-written object models for seven prompts as gold-standard object models. Next, we propose a similarity metric that quantifies the closeness between two object models. Finally, we define a scoring function that measures how well a generated object model fits a particular prompt.
3. We present three approaches for tackling the object model synthesis problem. The key innovation in all these algorithms is the use of GPT-3, a large language model which has background knowledge about the world. The first is a zero-shot approach, in which we directly ask GPT-3 what objects and fields a desired application has. The second is a one-shot approach, where we give GPT-3 an example of a successfully synthesized object model and ask it to generate an object model for a different prompt. The final is a more structured hierarchical approach, starting from the objects in the initial prompt and recursively prompting GPT-3 for related objects and fields. For most prompts, we observe that all three proposed approaches generate object models at a level similar to user-generated models and perform much better than randomly sampled models of different prompts.

2 RELATED WORK

Program synthesis: The field of program synthesis has had a long history, with a variety of approaches summarized in the survey (Gulwani et al., 2017). The first line of approaches to appear mostly focused on inductive synthesis (matching a set of input-output examples) approaches such as bottom-up search (Alur et al., 2015), top-down search (Feser et al., 2015), type-directed search (Osera & Zdancewic, 2015), and constraint-solving (Singh & Solar-Lezama, 2016). Later, however, richer forms of program specifications were used for synthesis.

In recent years, with new developments in machine learning, there have been more and more works exploring the potential of augmenting traditional synthesis techniques with neural networks; (Chaudhuri et al., 2021) provides a complete survey. These include approaches to learn abstractions and libraries from scratch (Ellis et al., 2020; Wong et al., 2021; Nye et al., 2020b), execution-guided approaches that evaluate partial program states (Nye et al., 2020a; Gupta et al., 2020; Chen et al., 2018), and approaches guided by natural language information (Wong et al., 2021; Ye et al., 2020b;a; Nye et al., 2019; Polosukhin & Skidanov, 2018).

Ontologies and Knowledge Graphs: There has also been a body of work that aims to build ontologies and knowledge graphs of natural language concepts, such as Yago (Suchanek et al., 2007), WordNet (Miller, 1995), and DBpedia (Auer et al., 2007). While these knowledge graphs have been applied in traditional NLP tasks such as question answering (Boiński et al., 2020), they are unable to provide specific insights for our synthesis task such as synthesizing fields for a certain object. As an example, when searching for nearest neighbors related to *student*, WordNet comes up with synonyms such as *pupil*, *educatee*, and *scholar*, while Yago provides a Wikipedia page for a student, a definiton of a student in Spanish, and an image containing many students. In addition, our synthesis task is very contextual: the fields of a student object would be very different if we were designing an app for teachers to manage the classroom vs a social app for students to make friends with one another. It is difficult to capture this form of context via ontologies and knowledge graphs.

Large Language Models: Recent years has also seen the birth of new works leveraging large language models (LLMs) like GPT-3 (Brown et al., 2020) to perform program synthesis. A few months ago, GitHub released a powerful code autocompletion tool called *GitHub Copilot* which uses context such as natural language comments and previous code in order to generate code. Copilot is built off of OpenAI's powerful machine learning model Codex (Chen et al., 2021), which translates natural language to code in almost a dozen programming languages. CodeBERT (Feng et al., 2020) learns representations of code and natural language for downstream tasks like code search and code documentation generation. Heyman et al. (2021) use GPT-2 trained on a corpus of well-documented and commented code to synthesize programs for data science and machine learning. Building off of LLMs, Austin et al. (2021) incorporate human feedback to repair generated code.

There have been other works combining traditional program synthesis techniques with large language models. Verbruggen et al. (2021) uses traditional inductive synthesis techniques with GPT-3 to learn small intermediate functions that cannot be represented symbolically. Jigsaw (Jain et al., 2021) uses LLMs to synthesize code but use program analysis techniques to do post-processing. Rahmani et al. (2021) take a component-based synthesis approach guided by LLMs which, for example, help rank candidate programs.

3 THE SYSTEM

Problem Definition and Notation: We begin by formalizing the object model generation task. The input to this task is a natural language specification of an application, e.g. *"I want a classroom management app that tracks students, the assignments they've submitted, and the grades they've earned on those assignments."* The goal is to generate a realistic object model of this application, where each object consists of a name and a set of fields. Formally, an object model consists of n objects $o_1, o_2, \cdots, o_n$. Each object o_i consists of a pair $(name_i, f_i)$ consisting of an object name and a set of m_i field names $f_i \triangleq \{f_i^j\}_{j=1}^{m_i}$. For example, the *student* and *submission* objects from Figure 1 might be represented as follows:

$$o_1 = (\textit{student}, \{\textit{name, email address, teacher, grade, assignments, submissions}\})$$
$$o_2 = (\textit{submission}, \{\textit{student, assignment, grade}\})$$

For all of our algorithms, our system uses the GPT-3 Q&A API with the text-davinci-001 model and default parameters from the OpenAI web API[1]: a temperature of 0 (greedy decoding), frequency penalty of 0, and presence penalty of 0.

Zero-shot GPT Generation: Our first approach is a zero-shot approach, in which the system prompts GPT-3 with our input specification and directly asks what objects and fields the desired application has (when evaluating this method, we manually parse the GPT-3 output into its objects and fields). A sample of the input and output of our zero-shot approach is shown in Figure 2.

One-shot GPT Generation: Our second approach is a one-shot approach, in which the system prompts GPT-3 with a complete object model for a classroom management app and asks it to generate an object model for a new prompt. In Figure 3, we show the input and outputs of GPT-3. The tables and fields are explicitly provided so that GPT-3's few-shot generation system learns a desired format. In practice, GPT outputs always conform to this specification format, so we write a parser that automatically convert these outputs into explicit object models.

Hierarchical GPT generation: The final approach we use is a hierarchical approach. At a high level, our algorithm follows an iterative depth first search, keeping a stack of all the objects we've

---

[1] https://beta.openai.com/playground

Input: *“I want a pet store app tracking customers. The pet store app should also keep the store’s inventory of pets and allow customers to buy and sell pets.*

Q: What objects and fields does this application have?”

GPT-3 Output: *“A: The application has the following objects: customer, pet, and inventory. The customer object has the following fields: name, address, and phone number. The pet object has the following fields: name, type, and description. The inventory object has the following fields: type, quantity, and price.”*

Figure 2: Sample input and output for our zero-shot object model generation algorithm

Input: *Q: Design a schema for the following prompt: ”I want a classroom management app that tracks students, the assignments they’ve submitted, and the grades they’ve earned on those assignments.*

A: Table: assignment
Fields: title (string), description (string), due date (datetime)

Table: student
Fields: name (string), email address (string), phone number (string), teacher (teacher), grade (int), assignments (List[assignment]), submissions (List[submission])

Table: teacher
Fields: name (string), students (List[student]), submissions (List[submission])

Table: submission
Fields: student (student), assignment (assignment), grade (int)

Q: Design a schema for the following prompt: ”I want a pet store app tracking customers. The pet store app should also keep the store’s inventory of pets and allow customers to buy and sell pets.”

GPT-3 Output:
A: Table: customer
Fields: name (string), email address (string), phone number (string), pet (pet), quantity (int), price (float)

Table: pet
Fields: name (string), type (string), description (string), quantity (int), price (float)

Table: inventory
Fields: pet (pet), quantity (int), price (float)

Figure 3: Sample input and output for our one-shot object model generation algorithm

encountered so far. We begin by extracting all the noun phrases from the prompt using a purely symbolic approach leveraging Stanford’s CoreNLP parser (Manning et al., 2014), which we denote by `extract_noun_phrases(prompt)`. We treat these noun phrases as a set of initial objects. From these initial objects, we recursively get a set of fields for each object, denoted by `extract_object_fields(object)`. For each of these fields, which we consider as new potential objects, we first check if we have encountered the field before. If not, we then test to see if the field should additionally be treated as a new object, and if so, add it to the stack. We do this via a `terminal(new_object)` check to ensure that we are only adding objects to the stack. In pseudocode, our algorithm is as follows:

```
def generate_object_model(prompt):
  initial_objects = extract_noun_phrases(prompt)
  stack = initial_objects
```

```
object_fields = {}
visited = {}
while stack is not empty:
  object = stack.pop()
  visited[object] = true
  object_fields[object] = extract_object_fields(object)
  for new_object in object_fields[object]:
    if visited[new_object] = false and not terminal(new_object):
      stack.append(new_object)
  return object_fields
```

Extracting Object Fields: Now, we explain `extract_object_fields(object)`. Our key insight is that GPT-3 can generate these relevant objects and fields from its domain knowledge. Specifically, given an object, this function asks GPT-3 what fields the object has via questions in the form *"Q: What attributes does a classroom management application have?"*. Remarkably, it would answer *"A classroom management app has a list of students, a list of assignments, and a list of grades."*. Next, the system parses this response to understand that a classroom management application object has *students*, *assignments*, and *grades* as fields. In order to avoid duplicate objects, we also ensure all object names are in singular form when prompting, like *student* instead of *students*.

Checking for Terminal Objects: The `terminal(object)` function helps the system discriminate which fields should be further treated as objects. While a *student* having a *name* is important, a *name* object having a list of characters is irrelevant. We once again use GPT-3 by prompting it with Q&A pairs like *"Q: What does a grade have?"* and *"A: A grade has nothing because it is an integer."*. The full prompt is deferred to Appendix B. This prompting mechanism helps GPT-3 learn that some objects have no fields, and the `terminal` function returns true if GPT says the object has nothing.

4 EVALUATION

We also define an evaluation procedure to quantify the correctness of a generated object model. This is tricky because for a single natural language description, there are many object models fitting the given description. First, in Section 4.1, we describe a data collection process to obtain gold standard models. Then, in Section 4.2, we present our full evaluation and scoring metric using these models.

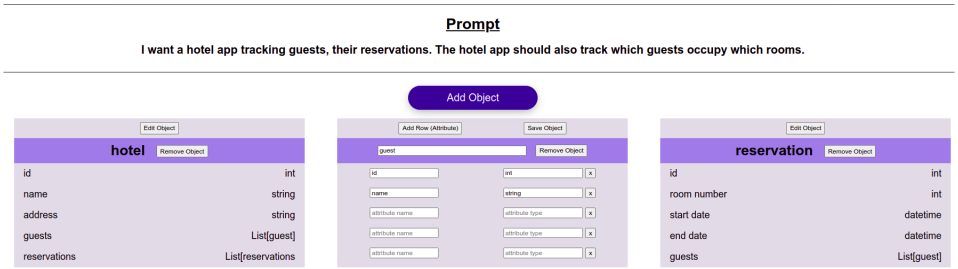
4.1 DATA COLLECTION

To obtain a set of gold-standard models, we set up an experimental testbed, asking participants to design object models from seven prompts simulating the following applications: pet store, restaurant, hotel, dating, library, company, and concert. The full prompts are in Appendix A. To help participants understand the task, we gave them an example of a full object model for a classroom application but emphasized that there is no correct answer. We included the example to demonstrate that participants should include objects and fields that weren't explicitly mentioned in the prompt.

As shown in Figure 4, the participant was shown a prompt at random and instructed to create an object model corresponding to the prompt. The users were given freedom over the number of objects they created and the number of fields they used for each object. For each field, they were asked to specify its name and its type. For types, we restricted participants to primitives, other objects they created, and lists of either. In total, we received 35 object models. However, we deemed 2 to be of extremely poor quality, leaving us with 33 gold-standard models.

4.2 SIMILARITY METRIC COMPONENTS

Now, we propose an evaluation metric to determine the similarity between two object models. Intuitively, our metric works as follows: consider two object models $O = \{o_1, \cdots, o_n\}$ and $O' = \{o'_1, \cdots, o'_{n'}\}$. For each object in O, say o_i, we find the most similar object in O', measured using an object similarity metric. We then average the similarities to get the overall object model similarity. To compute how similar two objects o_i and o'_j are, we employ a similar process: for each field f_i^m, we find the field in o'_j that is most similar to it via a field similarity metric, averaging the similarities to get the overall object similarity. Below, we make this concrete and precise.



Prompt

I want a hotel app tracking guests, their reservations. The hotel app should also track which guests occupy which rooms.

hotel	
id	int
name	string
address	string
guests	List[guest]
reservations	List[reservations

guest	
id	int
name	string
attribute name	attribute type
attribute name	attribute type
attribute name	attribute type

reservation	
id	int
room number	int
start date	datetime
end date	datetime
guests	List[guest]

Figure 4: The user interface for participants to design object models for our data collection effort.

Field similarity: To begin, we define the similarity between any two fields. When comparing two fields, it's important to consider the objects they were part of: many objects might have a "name" field that refer to very different things. Therefore, we concatenate the object name and field name, eg "pet name" or "customer name" when computing the similarity. Concretely, the similarity between fields f_i^a and f_j^b is defined as

$$\text{field\_sim}(f_i^a, f_j^b) = \text{phrase\_sim}(\text{concat}(name_i, f_i^a), \text{concat}(name_j, f_j^b)). \tag{1}$$

Here, phrase_sim is calculated by calling spacy's built in .similarity() method on two phrases, which is a word2vec-like similarity measure in $[0, 1]$, 1 indicating perfect similarity.

Object similarity: Next, we define a metric to evaluate object similarity. For two objects o_i, o_j to be similar, their object names and set of fields should both be similar. Overall, we define the object similarity as a combination f of the two, where we use $f(x, y) = xy$:

$$\text{o\_sim}(o_i, o_j) = f(\text{o\_name\_sim}(o_i, o_j), \text{o\_field\_sim}(o_i, o_j)). \tag{2}$$

First, we define the object name similarity similar to the field similarity, as

$$\text{o\_name\_sim}(o_i, o_j) = \text{phrase\_sim}(name_i, name_j). \tag{3}$$

Next, we define the object field similarity. Intuitively, if two objects o_i and o_j have identical sets of fields, there is a one-to-one mapping between each field in o_i and each field in o_j. If they aren't identical but similar, each field f_i^a should still have fields that are roughly similar to fields in o_j. For example, o_i might have a "name" field, but o_j might have "first name" and "last name" as fields. In this case, there would still be an approximate mapping between fields. We use this idea to define our metric: for each field f_i^a in object o_i, we compute the maximum field similarity between f_i^a and a field f_j^b in object o_j. We then average them to define

$$\text{o\_field\_sim'}(o_i, o_j) = \frac{1}{m_i} \sum_{a=1}^{m_i} \max_{b \in [m_j]} \text{field\_sim}(f_i^a, f_i^b). \tag{4}$$

Observe that if the fields in object i are a strict subset of the fields in object j, then $\text{obj\_field\_sim'}(o_i, o_j) = 1$, since each field in object i has a perfect match in object j. However, $\text{o\_field\_sim'}(o_j, o_i)$ will be smaller, since not every field in object j has a match in object i. Therefore, to make our metric symmetric, we define the overall field similarity as

$$\text{o\_field\_sim}(o_i, o_j) = \frac{1}{2}(\text{o\_field\_sim'}(o_i, o_j) + \text{o\_field\_sim'}(o_j, o_i)). \tag{5}$$

Object Model Similarity: Finally, we define a similarity function to evaluate how similar two full object-oriented models are. Consider two object models $O = \{o_i\}_{i=1}^{n}$ and $O' = \{o'_i\}_{i=1}^{n'}$. Similar to our object similarity metric, we average the similarities from each object $o_i \in O$ to its most similar object in O'. We define

$$\text{om\_sim'}(O, O') = \frac{1}{n} \sum_{i=1}^{n} \max_{j \in [n']} \text{o\_sim}(o_i, o_j), \tag{6}$$

$$\text{om\_sim}(O, O') = \frac{1}{2}\left(\text{om\_sim'}(O, O') + \text{om\_sim'}(O', O)\right). \tag{7}$$

Object Model Score: Finally, in order to evaluate an object model for a given prompt, we define a scoring metric taking object models to $[0, 1]$. Consider an object model O for a prompt P for which we wish to evaluate. Let $O_1, \cdots, O_k$ be the gold-standard object models for P. Intuitively, an object should have a high score if it is similar to any of the gold-standard models and have a low score otherwise. Therefore, we define the score of an object model on a given prompt to be its similarity to the most similar gold-standard model. Precisely,

$$\mathrm{score}_P(O) = \max_{i \in [k]} \mathrm{om\_sim}(O, O_i). \tag{8}$$

A more detailed motivation and evaluation of this metric is provided in Appendix C. As we will show in 1, the score function defined above effectively discriminates between human-generated object models, assigning high scores to human gold-standard models for the same domain, and low scores to object model pair from different domains.

5 RESULTS AND DISCUSSION

5.1 QUANTITATIVE RESULTS

We present the scores of each of the generated object models for all seven prompts in Table 1 when compared against our gold-standard benchmark. The *hier*, *1-shot*, and *0-shot* columns represent the algorithms in Section 3. The *similar* column of a prompt P measures the average score of gold-standard object models for that prompt when compared against the other gold-standard models. When calculating the score for an object O_i, we remove O_i from the set of gold-standard models to obtain a leave-one-out score because otherwise all the scores would be 1. The *different* column measures the average score of gold-standard object models in prompt P when compared against gold-standard models for the other prompts $P' \neq P$. This measures the similarity between models of a given prompt to gold-standard object models of other prompts. As we expect, these scores are much lower, as object models of different prompts should be different.

prompt	**hier**	**1-shot**	**0-shot**	**similar**	**different**
company	0.78	0.80	**0.84**	0.67	0.23
pet store	**0.81**	0.79	0.69	0.83	0.27
restaurant	0.86	**0.89**	0.77	0.81	0.31
hotel	0.81	0.88	**0.90**	0.92	0.21
dating	0.72	**0.83**	0.74	0.88	0.16
library	0.77	**0.79**	0.75	0.52	0.25
concert	0.75	**0.80**	0.68	0.76	0.26
average	*0.78*	***0.83***	*0.77*	*0.77*	*0.24*

Table 1: Scores for each of the three object model generation algorithms

5.2 QUALITATIVE RESULTS

In Figure 5, we show the three object models for the prompt *"I want a pet store app tracking customers. The pet store app should also keep the store's inventory of pets and allow customers to buy and sell pets."* We include our full list of generated schemas for other prompts in Appendix D. Each of the methods has its own merits and flaws. First, we notice that all three methods are able to generate relevant information that isn't explicitly provided in the prompt (such as a customer's name). Second, in the hierarchical table, we can see that each field is either a primitive (such as name, address, phone number) or is the name of another object. However, in the 0-shot and 1-shot table, there are tables that are never referenced, like *inventory* in the 1-shot case and *purchase* in the 0-shot case. Third, we note that in contrast to the other methods, the 1-shot method often generates a more exhaustive list of fields.

6 CONCLUSION AND FUTURE WORK

In this work, we introduced a new class of important program synthesis problems known as *specification reification*, focused on incorporating domain knowledge into traditional program synthesis.

Hierarchical (Score: 0.81)

customer	store	pet	pet store app	purchase
- name - email address - phone number - pet - purchase	- name - address - pet - purchase	- name - description - price - purchase	- customer - pet - purchase - store	- customer - pet - price

1-shot (Score: 0.79)

customer	pet	inventory
- name - email address - phone number - pet - purchased - sold	- name - type - description - age - gender - purchased - sold	- pet - quantity - price

0-shot (Score: 0.69)

customer	pet	inventory	purchase
- first name - last name - email - phone	- name - type - age - gender	- type - age - gender	- customer - pet - inventory

Figure 5: Hierarchical, 1-shot, and 0-shot object models for a pet store application

First, we presented one specific instance of this task, object model synthesis, and designed a metric to evaluate performance on this task. Then, we demonstrated three different algorithms to solve this task, showing that we are able to synthesize, to some extent, object models satisfying the specification. These object models include fields and tables not explicitly mentioned in the original prompt. Evaluating these algorithms on our metric is a first step towards showing that our generated object models are similar to human gold-standard models.

We believe that specification reification is an important problem and welcome researchers to introduce other synthesis problems fitting this framework. We identify many attractive directions for future investigation: first, there is significant room for the discovery novel approaches that improve upon the three we present. Second, we encourage a more rigorous data collection effort and investigation of our evaluation procedure: the quantity of object models we collected is likely too small to fully cover the set of possible gold-standard models. Third, GPT often gives misleading or incorrect information, so one way of correcting this is to explore how human interactivity can be incorporated to correct and augment the synthesized object model. Finally, since software engineers often spend time designing what methods to implement, it would be interesting to extend the object model to include these method names, types, and descriptions, e.g., turning in assignments for a classroom application.

REFERENCES

Rajeev Alur, Pavol Černỳ, and Arjun Radhakrishna. Synthesis through unification. In *International Conference on Computer Aided Verification*, pp. 163–179. Springer, 2015.

Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pp. 722–735. Springer, 2007.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Tomasz Boiński, Julian Szymański, Bartłomiej Dudek, Paweł Zalewski, Szymon Dompke, and Maria Czarnecka. Nlp questions answering using dbpedia and yago. *Vietnam Journal of Computer Science*, 7(04):339–354, 2020.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, Yisong Yue, et al. *Neurosymbolic Programming*. Now Publishers, 2021.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

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

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*, 2020.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices*, 50(6):229–239, 2015.

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

Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. Synthesize, execute and debug: Learning to repair for neural program synthesis. *Advances in Neural Information Processing Systems*, 33:17685–17695, 2020.

Geert Heyman, Rafael Huysegems, Pascal Justen, and Tom Van Cutsem. Natural language-guided programming. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 39–55, 2021.

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. *arXiv preprint arXiv:2112.02969*, 2021.

Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pp. 55–60, 2014.

George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11): 39–41, 1995.

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

Maxwell Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B Tenenbaum, and Armando Solar-Lezama. Representing partial programs with blended abstract semantics. *arXiv preprint arXiv:2012.12964*, 2020a.

Maxwell Nye, Armando Solar-Lezama, Josh Tenenbaum, and Brenden M Lake. Learning compositional rules via neural program synthesis. *Advances in Neural Information Processing Systems*, 33:10832–10842, 2020b.

Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices*, 50(6):619–630, 2015.

Illia Polosukhin and Alexander Skidanov. Neural program search: Solving data processing tasks from description and examples. 2018.

Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. Multi-modal program inference: a marriage of pre-trainedlanguage models and component-based synthesis. *arXiv preprint arXiv:2109.02445*, 2021.

Rohit Singh and Armando Solar-Lezama. Swapper: A framework for automatic generation of formula simplifiers based on conditional rewrite rules. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 185–192. IEEE, 2016.

Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pp. 697–706, 2007.

Gust Verbruggen, Vu Le, and Sumit Gulwani. Semantic programming by example with pre-trained models. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–25, 2021.

Catherine Wong, Kevin M Ellis, Joshua Tenenbaum, and Jacob Andreas. Leveraging language to learn program abstractions and search heuristics. In *International Conference on Machine Learning*, pp. 11193–11204. PMLR, 2021.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Benchmarking multimodal regex synthesis with complex structures. *arXiv preprint arXiv:2005.00663*, 2020a.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Optimal neural program synthesis from multimodal specifications. *arXiv preprint arXiv:2010.01678*, 2020b.

A PROMPTS

We used the following seven prompts throughout:

- I want a company management app where I can track my employees, their schedules, and their compensation.
- I want a pet store app tracking customers. The pet store app should also keep the store's inventory of pets and allow customers to buy and sell pets.
- I want a restaurant management app tracking customers, their reservations, their orders, and menu items.
- I want a hotel app tracking guests, their reservations. The hotel app should also track which guests occupy which rooms.
- I want a dating app where students can indicate other students they are interested in, and the dating app will match them.
- I want a library app to track books. The customers can borrow books and return books from the library.
- I want a concert management app that lists concerts, their dates, and their locations. The customers can also buy tickets to concerts.

B 1-SHOT GPT PROMPT

Below, we include the full prompt we provide to GPT-3 in the 1-shot setting:

I want a classroom management app that tracks students, the assignments they've submitted, and the grades they've earned on those assignments.

Q: What does a classroom management app have?

A: A classroom management app has a list of students, a list of assignments, and a list of grades.

Q: What does an assignment have?

A: An assignment has a title, a description, a due date, and a list of grades.

Q: What does a student have?

A: A student has a name, an email address, a phone number, a list of assignments, and a list of grades.

Q: What does a grade have?

A: A grade has nothing because it is an integer.

Q: What does a title have?

A: A title has nothing because it is a string.

Q: What does a description have?

A: A description has nothing because it is a string.

Q: What does a due date have?

A: A due date has nothing because it is a string.

Q: What does a name have?

A: A name has nothing because it is a string.

Q: What does a email address have?

A: An email address has nothing because it is a string.

Q: What does a phone number have?

A: A phone number has nothing because it is a string.

C META-METRIC

In Section 4.2, we defined a framework to capture the similarity between two object models. Since such a metric has not been established before, it is important to ensure that the metric faithfully captures the similarity between two object models. Therefore, we define a meta-metric to capture the faithfulness of a metric. The methodology in this section can be applied to future work in other instances of specification reification.

Consider two different prompts P and Q. An important desideratum for an evaluation metric is that the similarity between object models for the same prompt should be closer than object models for different prompts. To that extent, let $O_1^P, \cdots, O_k^P$ be gold-standard object models for prompt P and $O_1^Q, \cdots, O_l^Q$ be gold-standard object models for prompt Q. For a given similarity metric, we define

$$\mu_{PQ} = \frac{1}{2k} \sum_{i \in [k]} \mu_Q(O_i^P) + \frac{1}{2l} \sum_{i \in [l]} \text{score}_P(O_i^Q),$$

$$\mu_{PP} = \frac{1}{k} \sum_{i \in [k]} \text{score}_P(O_i^P), \mu_{QQ} = \frac{1}{l} \sum_{i \in [l]} \text{score}_Q(O_i^Q)$$

In a similar spirit to contrastive learning, we want μ_{PP} and μ_{QQ} to be close to 1, while we want μ_{PQ} to be close to 0. This motivates the meta-metric $\frac{|\frac{\mu_{PP}+\mu_{QQ}}{2} - \mu_{PQ}|}{1+|\mu_{PP}-\mu_{QQ}|}$ which takes values in $[0, 1]$. In the best case scenario where $\mu_{PP} = \mu_{QQ} = 1$ and $\mu_{PQ} = 0$, the metric has a value of 1. In the worst case scenario where $\mu_{PP} = \mu_{QQ} = \mu_{PQ}$, signifying that the metric cannot differentiate between object models of different prompts, the metric has a value of 0.

Recall that in Eq. 2, the function $f(\cdot, \cdot)$ calculates the similarity between two objects as a combination of the object names and object fields. We considered two ways to combine: $f(x, y) = xy$, capturing the fact that object names and object fields should be similar, and $f(x, y) = \lambda x + (1 - \lambda)y$, representing a relative weighting between the two aspects. Apart from f, we also considered replacing the $\max$ in Eq. 4.2 with an average.

In Table 2, we show the different values of the meta-metric for various combinations of λ, f, and aggregation strategy. In the f column, "ws" represents the weighted sum combination $f(x, y) = \lambda x + (1 - \lambda)y$, while "prod" represents the product $f(x, y) = xy$. In the "agg" column, $\max$ represents using $\text{score}_P(O) = \max_{i \in [k]} \text{om\_sim}(O, O_i)$, while avg represents using $\text{score}_P(O) = \frac{1}{k} \sum_{i \in [k]} \text{om\_sim}(O, O_i)$. We found that using $f(x, y) = xy$ and $\max$ aggregation worked best.

λ	f	agg	score
0	ws	max	0.25
0.1	ws	max	0.27
0.2	ws	max	0.29
0.3	ws	max	0.31
0.4	ws	max	0.34
0.5	ws	max	0.36
0.6	ws	max	0.38
0.7	ws	max	0.39
0.8	ws	max	0.41
0.9	ws	max	0.43
1	ws	max	0.45
0	ws	avg	0.24
0.1	ws	avg	0.26
0.2	ws	avg	0.28
0.3	ws	avg	0.30
0.4	ws	avg	0.31
0.5	ws	avg	0.33
0.6	ws	avg	0.35
0.7	ws	avg	0.36
0.8	ws	avg	0.38
0.9	ws	avg	0.40
1	ws	avg	0.41
N/A	prod	max	**0.46**
N/A	prod	avg	0.41

Table 2: The meta-metric score (0–1, with 1 being the best) for different possible similarity metrics

D SAMPLES OF GENERATED SCHEMAS

In this section, we provide a full list of GPT generated schemas.

D.1 OBJECT MODELS FOR A DATING APPLICATION

Prompt: I want a dating app where students can indicate other students they are interested in, and the dating app will match them.

Hierarchical (Score: 0.72)

dating app	student	other student
- student - students they are interested in - students they are matched with	- name - email address - phone number - students they are interested in - students they are matched with	- name - email address - phone number - students they are interested in - students they are matched with

students they are interested in	students they are matched with
- name - email address - phone number - students they are matched with	- name - email address - phone number - students they are matched with

1-shot (Score: 0.83)

student	match
- name - email address - phone number - gender - grade - interested_in	- student1 - student2 - grade - date

0-shot (Score: 0.74)

student	match	date
- firstName - lastName - email - yearInSchool	- student 1 - student 2 - date	- date - time - location

Figure 6: Hierarchical, 1-shot, and 0-shot object models for a dating application

D.2 OBJECT MODELS FOR A CONCERT MANAGEMENT APPLICATION

Prompt: I want a concert management app that lists concerts, their dates, and their locations. The customers can also buy tickets to concerts.

Hierarchical (Score: 0.75)

customer	concert management app	ticket	concert
- name - email address - phone number - concert	- concert - customer - ticket	- number - customer - date - location	- name - date - location - ticket

1-shot (Score: 0.80)

concert	customer	ticket
- name - date - location - price	- name - email address - phone number - address - city - state - country	- customer - concert - price - quantity

0-shot (Score: 0.68)

concert	date	location
- name - date - location - tickets	- date - time - location	- name - address - city - state - zip

Figure 7: Hierarchical, 1-shot, and 0-shot object models for a concert management application

D.3 OBJECT MODELS FOR A LIBRARY APPLICATION

Prompt: I want a library app to track books. The customers can borrow books and return books from the library.

Hierarchical (Score: 0.77)

library app	library	customer	book
- customer - book - books borrowed	- customer - book - books borrowed	- name - email address - phone number - books borrowed - books returned	- title - description - due date - books borrowed

1-shot (Score: 0.79)

customer	book	library
- name - email address - phone number - books	- title - author - pages - due date	- name - customers - books

0-shot (Score: 0.75)

customer	book	book loan
- name - address - phone number	- title - author - ISBN	- customer - book - due date

Figure 8: Hierarchical, 1-shot, and 0-shot object models for a library application

D.4 OBJECT MODELS FOR A HOTEL APPLICATION

Prompt: I want a hotel app tracking guests, their reservations. The hotel app should also track which guests occupy which rooms.

Hierarchical (Score: 0.81)

reservation	hotel app	guest	room	date
- guest - room - date - date	- guest - reservation - room	- name - email address - phone number - reservation - room	- number - description - guest - reservation	- year - month - day

1-shot (Score: 0.88)

guest	reservation	room
- name - email address - phone number - room number - check-in date - check-out date	- guest - room number - check-in date - check-out date - status	- number - description - guests

0-shot (Score: 0.90)

guest	reservation	room
- name - email - phone number	- date - time - room	- number - name

Figure 9: Hierarchical, 1-shot, and 0-shot object models for a hotel application

D.5 OBJECT MODELS FOR A RESTAURANT APPLICATION

Prompt: I want a restaurant management app tracking customers, their reservations, their orders, and menu items.

Hierarchical (Score: 0.86)

reservation	customer	restaurant management app	order	menu item
- date - time - table number - customer	- name - phone number - reservation - order	- customer - reservation - order - menu item	- date - time - table number - menu item - customer	- name - description - price - order

1-shot (Score: 0.89)

customer	reservation	order	menu item
- name - email address - phone number - reservation date - reservation time - table number - order date - order time - menu item - quantity	- customer - reservation date - reservation time - table number	- customer - order date - order time - menu item - quantity - price	- name - description - price

0-shot (Score: 0.77)

customer	reservation	menu
- first name - last name - email - phone number - address - city - state - zip code	- date - time - number of people - menu items - notes	- name - description - price - quantity

Figure 10: Hierarchical, 1-shot, and 0-shot object models for a restaurant application

D.6 OBJECT MODELS FOR A PET STORE APPLICATION

Prompt: I want a pet store app tracking customers. The pet store app should also keep the store's inventory of pets and allow customers to buy and sell pets.

Hierarchical (Score: 0.81)

customer	store	pet	pet store app	purchase
- name - email address - phone number - pet - purchase	- name - address - pet - purchase	- name - description - price - purchase	- customer - pet - purchase - store	- customer - pet - price

1-shot (Score: 0.79)

customer	pet	inventory
- name - email address - phone number - pet - purchased - sold	- name - type - description - age - gender - purchased - sold	- pet - quantity - price

0-shot (Score: 0.69)

customer	pet	inventory	purchase
- first name - last name - email - phone	- name - type - age - gender	- type - age - gender	- customer - pet - inventory

Figure 11: Hierarchical, 1-shot, and 0-shot object models for a pet store application

D.7 OBJECT MODELS FOR A COMPANY MANAGEMENT APPLICATION

Prompt: I want a company management app where I can track my employees, their schedules, and their compensation.

Hierarchical (Score: 0.78)

schedule	employee	company management app
- title - description - start date - end date	- name - email address - phone number - schedule - compensation	- employee - schedule - compensation

1-shot (Score: 0.80)

employee	schedule	pay
- name - position - schedule - pay	- day - time - position	- day - position - hours

0-shot (Score: 0.84)

employee	schedule	compensation
- name - contact info - schedule - compensation	- date - time - location	- type - amount - date

Figure 12: Hierarchical, 1-shot, and 0-shot object models for a company management application