DRAFT: Programming with Non-algorithmic Specifications

Extracting Correctness From LLMs

Daniel Patterson[a], Smaran Teja[a], Lukas Tegge[a], and Rachel Chen[a]

a Northeastern University, Boston, USA

Abstract AI-based tools have promised to revolutionize, or eliminate, programming. And yet, all current state-of-the-art tools suffer from a fundamental weakness: whether autocompletion or chat-based, they rely upon programmers to determine if the code produced is actually correct. Not only is reading code difficult, but this task is compounded by the psychological phenomenon of "automation bias": that humans put more trust in automated systems than is warranted. While many have studied this central problem of validation, few have proposed alternatives that avoid this inherent issue.

In this paper, we present a vision of a better way for AI to be incorporated into programming without compromising reliability, correctness, or security. In our view, programmers should write down executable specifications, and the AI-based tools should be used, in the background, to find conformant solutions that are validated against the specifications. This differs from typical use in, e.g., autocomplete based systems, in that the specifications are written entirely concretely, with no aid, and yet implementations are generated entirely automatically, with no oversight by the programmer.

Our view is that Large-Language Models (LLMs) that power modern AI tools should be used to build compilers from non-algorithmic specifications to code, but as with any compiler, we need some way of assuring correctness. To do this, we require the specifications be validated automatically: in this paper, our prototype tool Synth does this by expecting specifications in the form of test cases. In case of test failures, the models are automatically re-queried for an updated solution without programmer intervention.

We approach the task of extracting correct solutions from LLMs from an understanding that programs can be decomposed into two parts. The first part is inherently creative, or determined only by the problem domain: these details cannot be figured out by a language model, no matter how sophisticated, because they come from outside the program. These form the specification, but can be expressed non-algorithmically, capturing the intended behavior in whatever way makes sense. The second part of the program is the algorithmic code that realizes this behavior. At least in principle, this should be synthesizable, since it is mostly or entirely determined from the specification.

We present this vision for the future in the form of a concrete tool Synth, which both serves to illustrate our idea but also as a prototype that can be used today. In doing so, it can lead the way towards a future where LLMs are integrated into programming tools without compromising reliability, maintainability, or security.

ACM CCS 2012

- **Software and its engineering** → **General programming languages**; **Software verification and validation**;

Keywords programming, llm, specification

The Art, Science, and Engineering of Programming

Perspective The Art of Programming

Area of Submission Program verification, Programming environments, General-purpose programming

Part of the title comes from a phrase from David Parnas's prescient essay [20] about the limitations of software engineering, in which he discusses, in the context of AI and automatic programming, how miraculous new technologies often become ordinary in hindsight, rather than fundamentally changing the process of software construction.

1 Introduction

Countless tools promise to improve programmer productivity using AI-based Large Language Models (LLMs), from GitHub/Microsoft's Copilot [16] and similar auto-completion engines [1] to chat-based interfaces like OpenAI's ChatGPT [19], sometimes fine tuned to programming.[2] All of these tools suffer from the same fundamental weakness: they rely upon programmers to determine whether the code produced by the tool is suitable for the task, a task that programmers often struggle with [12].

We can understand this fundamental limitation from multiple directions, all serious. One issue that is well-known and studied is a phenomenon called "automation bias": that humans suffer from overblown trust of automated systems [24]. This means they attribute expertise where it may not be warranted, and that, for example, the marketing of many of these tools as a replacement for a pair programmer is misleading at best. When working with another human, people will interrogate the code proposed by the other person in ways that they would not when presented with the same code by an automated assistant.

Another issue, which interacts with the first, is that reading programs is difficult. It consumes a significant portion of time even under normal circumstances [17, 28], with coherent code that already existed in the codebase, was likely reviewed, etc.

Perhaps obvious, but with either code completion or chat-based models, the only validation that code gets before being inserted into programs is a review by the programmer, likely distorted by their own automation bias.

In this paper, we propose a different approach. If we can understand code synthesis as a process of translating (or compiling) a non-algorithmic specification (in the form of text, code, etc) into an implementation, then the challenge is knowing whether that compiler is correct. Just as we should not expect programmers to validate the output of their compilers by hand, expecting programmers to accurately validate the correctness of these tools is unrealistic. So what else can we do?

In order to understand this, we need to be more careful about what parts of the program can (or should) be synthesized, and what parts must be written by the programmer. i.e., what is the input to our compiler, rather than the output.

Consider a structured approach to programming that is used for teaching students [9]. Since LLMs purport to fundamentally change how programming works, it is worthwhile to begin with a systematic approach to programming, and see how (or

[1] e.g., Salesforce's CodeT5 [26], Hugging Face's SafeCoder [11], Stability AI's StableCode [1], etc

[2] e.g., Meta's Code Llama [15], etc

if) it should change with the introduction of these tools. The idea from [9], named the "Design Recipe", is a data-driven approach to programming that begins with specification and gradually refines into programs. It teaches students how to decompose problems into data, and how to design functions in a systematic manner, first by determining the data that the functions operate over (the "signature"), next by writing down a concise description of behavior (the "purpose statement"), third by writing examples of use (aka "test cases"), and only then by proceeding to the body of the function (the "implementation"). While the four steps are presented as sequential steps that student-programmers do in order, the first three steps always require some amount of *thought*, usually involving (perhaps iterative) interrogation of the problem at hand, whereas the fourth step ("implementation") is ideally quite constrained by those preceding it, such that it sometimes proceeds relatively mechanically. Sometimes, of course, issues arise in the implementation that require revisiting earlier steps, but one way of understanding these four steps is that the first three form a specification for the last.

In the setting of a single function, then, we can use this insight to alter how we utilize the LLM. In typical autocompletion-based systems, unstructured assistance is available at any point, as soon as programmers type a few characters. A better intervention is to not provide any assistance in steps 1-3 ("signature", "purpose", and "tests"), which are the *specification* and unlikely to be foreseeable, with sufficient precision, by an LLM. Instead, the model should focus exclusively on step 4, the implementation. More importantly, given that sufficiently detailed signature/tests should highly constrain possible implementations, we should not need to show the resulting code to the programmer at all, entirely sidestepping issues of automation bias, difficulty of code comprehension, etc. Instead, conformance with the test suite that the programmer wrote can be checked *automatically*. We thus advocate for less intervention of LLMs in the process of specification, which fundamentally is about precisely expressing intent that the language model cannot know, and more intervention of the LLM in the actual implementation once given the intent.

Moving across to the other half of the Design Recipe, which concerns designing data representations to model information. Data is designed in four steps as well. First by specifying what values make up data (the "definition"), second a descripton of how those values model the information at hand (the "interpretation"), third a collection of examples, and fourth a code snippit that shows how such data is typically used (the "template").

As with functions, we can see that the first three steps fundamentally involve human interrogation of a problem: how data should be represented, what it means, and demonstration of the utility of the definition via examples. But again, step 4 (the "template") is derivable from the first three, and in the context of an LLM-powered programming language, could be eliminated.

Considering programs that are larger than single data definitions or functions, we observe that the decomposition of problems into data, the determination of components, interfaces, system boundaries, etc, are likely creative endeavors that will resist attempts to automate. Indeed, the complexity that arises from this has been studied for many years [20].

The fact that the LLM-based tools that are proliferating are primarily completion-based is a testament to this: beyond a relatively small scale, the complexity spirals, and so they aim to insert themselves where they can construct small fragments of programs. But this is done without full acknowledgment that the bulk of "programming" is thus still in the hands of the programmer. And indeed, in work (e.g., [3, 23]) that aims to study how "real" (or non-expert) programmers use these tools, this is a central issue: LLMs work quite well on well-specified small fragments, but knowing both what and how to specify are core programming skills, sowing doubt in the idea that these tools are going to eliminate or even significantly change programming, at least any more than previous productivity increases (high-level programming languages, garbage collection, etc) have.

We present our alternative in the form of a tool, SYNTH, usable today, but primarily as a vision for the future. We have prototyped it both as researchers, and with students. Using it, programmers write code [3] as usual: creating modules, data definitions,[4] and functions, but any implementation that they wish to implement via the synthesis engine they can.[5]

In this way, we realize the distinction described above: that program decomposition, data design, signatures, purpose statements, and tests are fundamentally creative activities for people to do, but that the code that follows might be deferable to automation.

Specifically, the annotation that programmers add will cause the runtime to query an LLM [6] for a conformant implementation. The implementation will be validated against provided tests, and if it passes, will be used as the implementation. If the resulting code fails to pass tests, the synthesis engine will requery the LLM for a correction, using the failing test(s) as feedback. If after a configurable number of retries, no conformant implementation can be found, an error is reported.

There are a few emergent properties of this system that are worth dwelling on. First, the obvious criticism: what if the tests underconstrain the behavior, and the generated code is "wrong"? To which we ask the reader: can the programmer, not having understood the problem well enough to put in sufficient tests, not implement an incorrect solution as well? Indeed, rather than trusting that a programmer will be able to identify violations of the specifications by reading code, we think they will have an easier time focusing their attention on the specification itself. This is, indeed, one of the key lessons of the Design Recipe, and the reason why example function invocations (tests) precede implementation: by focusing only on the input/output behavior, programmers will have an easier time specifying behavior than if they are also thinking about *how* such a function can be implemented. This is also the philosophy that drives Test Driven Development. In our implementation, no source code is provided for the generated implementations. Rather than reading code, we

[3] Using Python, given how well currently available models perform on it
[4] Classes, in Python
[5] Via a Python decorator `@synth`
[6] We use OpenAI's GPT-4-Turbo currently

suggest that programmers interrogate the behavior of the generated code: these investigations should then be reified into tests.[7]

Second, if a bug is identified, due to program behavior not being as expected, the correction is much more direct. Rather than a three step process: first identifying the buggy behavior (the function that does not behave as expected), second identifying how the code is wrong (what *in* the function is wrong), and third how to correct the behavior, our model allows the programmer to update the specification to include the refined understanding of the problem and defer reimplementation (including any cascading changes) to the synthesis runtime. More specifically, they add the regression test, and that's it.

While the tool SYNTH presented in the paper is, indeed, a prototype, it is a prototype that presents a vision of the future, derived from a structured understanding of how programs should be constructed. The principles that underly it derive from experience working with beginning programmers, and indeed, we have presented the tool to exactly that same population. The rest of the paper goes over, in detail, both how the system works, tradeoffs explored, experiments with it, and where we think this technology should go next.

2 Main Idea

Large-Language Model powered tools have promised to revolutionize programming. While some envision a future where the models execute directly [27], most imagine a mechanism where the models, which generate text, are used to generate code that is then reviewed by programmers before or as it is inserted into their programs. The quintessential example of this is GitHub CoPilot, called "Your AI pair programmer" [16], which primarily operates as block-level code completion within IDEs. While there are now more integrated ways to use the IDE to provide code completion, a typical use might be writing a documentation comment for a method and having an implementation filled out for you. Carefully hidden on the marketing pages [16] are caveats that it may make mistakes, and programmers are instructed to review the code to ensure that it does what they expect it to.

For certain repetitive code that is easily validated by skimming it, this may work out well. The problem, of course, is that simple repetitive code generally can already be abstracted through ordinary language mechanisms, so it seems unlikely to be the source of a revolutionary change. More realistically, LLM-based autocompletion must be used to implement non-trivial logic, for which quite careful review is necessary if correctness is desired. This is potentially a problem, as there is extensive research [4, 6, 8, 21] that suggests code review, while useful, is certainly not infallible, or necessarily even all that effective for certain types of bugs.

[7] This is easy to do in our system, given one form of specification, `doctest` tests, that we support are transcripts of interactive sessions.

We propose a different approach, based on considering that review process as the most important part of the entire process. Through reflection, we realize that provided the synthesized code *behaves* as expected, the actual text of the code doesn't matter. This is important, because it side-steps the (potentially fraught) issue of code review. Instead, the task becomes describing the behavior of the code that is desired. Now, an exact description of the behavior is likely indistinguishable in complexity from the original code, but an *approximate* specification of behavior is already something incorporated into most software development processes in the form of test suites. Indeed, many software engineers advocated writing tests before implementations in order to figure out architectural mistakes before committing to the effort of implementation.

This is the core idea: write executable specifications (in the form of test suites) and then use synthesis to find conformant implementations. By doing this, the "review" is done automatically, by running the specification against the synthesized code.

This has several interesting consequences:

1. While software engineering best practices recommend extensive testing, this is typically only enforced, if at all, by code review processes. In a system where the only code that is written down are test cases, test suites are likely to be much more extensive.
2. A benefit of this is that in case of buggy behavior, rather than first writing "regression" tests (that pinpoint where the code was not working) and then identifying how the code should change, the regression tests should be the only change needed, provided the synthesis is capable of creating a new conformant implementation.
3. Such changes can have cascading effects: i.e., if one function or class has a bug that necessitates a change, anything that depends upon it will be re-synthesized and the new implementation will be validated against the original specification.
4. While the decrease in effort vs. typical development is most extreme in case of fixing bugs, refactoring in general will result in some decrease in effort, as only the test suites need to be updated, rather than both test suites and implementations.

3 Architecture

We show the overall architecture of SYNTH in Figure 1. Fundamentally, there is a unit of synthesis, which in our implementation is either a function or a class. This unit has no implementation (if a function) or perhaps a partial implementation (if a class) in the program text, but at execution time, must have an implementation or else the program will be invalid. It is the job of SYNTH to find a conformant implementation. It does this by several steps.

3.1 Caching

First, we look for an existing implementation for the given function or class. The cache key used for this incorporates all aspects of the function/class relevant to synthesis,

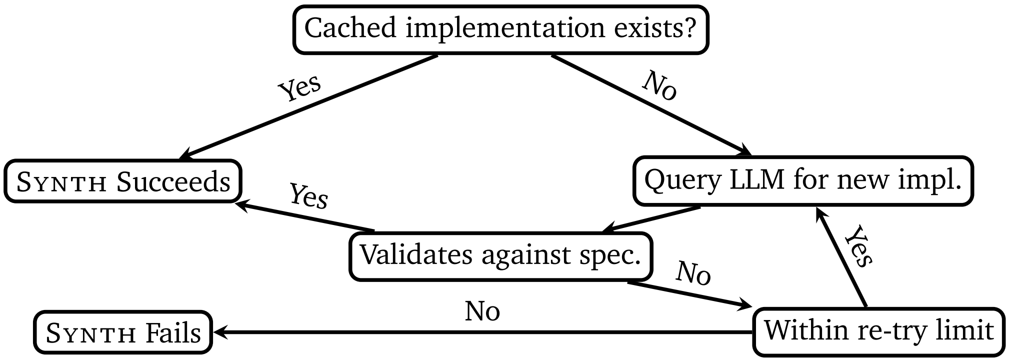

Figure 1 Architecture of SYNTH

as well as any dependencies that it has, so if tests are added, changed, or if any dependencies are changed in any way, the cache will be invalidated. In addition to this code, the cache key also incorporates the placeholder code, which includes the docstring, function / class / method names, etc.

Caching is a key aspect of the architecture because since the implementations are not stored in the file, performance would be abysmal if every time a file was loaded all implementations would need to be synthesized. A given unit might take several queries and a query might take several seconds, so loading a file with many units of synthesis could easily take dozens of seconds without caching. With caching, it is certainly slower than native compiled bytecode, but still only takes milliseconds.

3.2 Querying

Assuming a cached implementation does not exist, we must query the LLM for an implementation. We include several elements:

1. Standard boilerplate instructions, describing the task (code completion), fine tuning the response (asking it not to explain the code it produces, not to use libraries other than those explicitly allowed), etc.
2. The code to be completed: for a function, this is the function header and docstring. For a class, we provide the complete implementation that was written into code, as some methods may have been implemented by the programmer. Importantly, the docstring may include `doctest` tests, which is the form of unit tests that we currently support. In the future, we may add support for the `unittest` library.
3. All the dependencies: for functions/classes implemented in the same file, we provide the full source; for modules, we provide just the name for conciseness. This is a tradeoff where future work could explore better ways to provide parts of the rest of the codebase for the synthesis.
4. All property-based tests written using the Hypothesis [14] library that were tagged as relevant to this unit.

The query is constructed as a conversation, where every message back and forth is included so that in case there are errors or test failurues, subsequent re-queries include the full history for the LLM.

3.3 Validation

Once the model returns a response, we attempt to extract code from it, execute the code which should result in either a class or function. We then validate this new executable object against whatever specification that we have: unit tests that existed in the original docstring (*not* in the docstring that comes back from the LLM), as written by the programmer, and property based tests that were tagged as relevant to this implementation.

If the code fails to run, or if any of the tests either fail or raise errors, we construct a new message with the corresponding error, add it to the query, and re-prompt the model. This can happen up to a bounded number of times, as the models do take the errors into account and will often construct alternate implementations avoiding the bugs present in the original implementation. While we do provide all the tests we validate in the original query, we've noticed that, especially for more interesting examples where more sophisticated implementations are necessary, the model may initially attempt to provide a simpler (and therefore buggy) implementation, but may respond to the feedback provided.

3.4 Re-tries

The number of times the system re-tries is a somewhat important tricky optimization: in case that the specification is incomplete, encompasses too much functionality, or is simply contradictory, the LLM may *never* return a correct solution, so we want to bound the number of attempts. The failure to synthesize signals to the programmer that they need to address one of these points. On the other hand, the models do sometimes resolve issues with feedback (though, [7] shows that the current models don't do this particularly well), and so too few attempts can give false negative results that cause the programmer to do spurious extra work.

Our current system takes a relatively simple approach. For functions, we use a hardcoded default number of attempts, which can be changed by setting a parameter in the decorator. Classes use the same override for a base count, but increase the number of tries based on the number of methods. The idea is that there is some inherant complexity per method, so we are willing to give it a few more tries to get an entire class correct.

A more sophisticated system might use a heuristic to see if it seemed like the previous failed attempt was due to a problem that needed to be resolved by the programmer or was a failure of the model. Of course, figuring out such a heuristic might be quite difficult, hence our relatively simple approach.

4 Assessment

While SYNTH is certainly a prototype and a way for us to demonstrate our idea of what the future of AI-powered programming might look like, we did have a group of around 50 undergraduate students in an introductory programming class try using it in the context of a regularly scheduled programming lab.

They were given an example invocation of the tool, with a `Clock` class and a function `add_second` function that added a second, handling wraparound for minutes and seconds. In order to learn how to use the tool, we showed them how they could specify the behavior of that function using unit tests, and how to make the function depend on the class.

Then, the main task for the lab was to implement a simplified version of the card game "Black Jack" which had a fixed dealer hand and no betting. The students were provided data definitions as starter code:

```
@dataclass(frozen=True)
class Card:
  """
  Represents a card that has a suit and value
  """
  suit: str # H, C, S, D
  value: str # A, 2, 3, ... J, Q, K

@dataclass(frozen=True)
class BlackjackGame:
    """
    Represents the data that goes into a Black Jack game
    """
    player_cards : tuple = tuple()
    dealer_cards : tuple = (Card("H", "10"), Card("H", "7"))
    deck : tuple = ()
```

And were given as a prompt to implement a series of functions that allowed them to create a purely functional game, evaluated at the REPL as follows:

```
>>> gm = shuffle_deck(add_to_deck(BlackjackGame(), 52))
>>> gm1 = deal_initial(gm)
>>> render_player_hand(gm1)
'4S KS'
>>> gm2 = hit(gm1)
>>> render_player_hand(gm2)
'4S KS 2D'
>>> gm3 = hit(gm2)
>>> render_player_hand(gm3)
'4S KS 2D 10C'
>>> winner(gm3)
'dealer'
```

The choice to implement this in a dataclass/function only style was two-fold: first, the class was entirely taught using a purely functional language, so the transition to Python (which some had never seen before) for the lab was easier. Second, at the point when the lab was run, Synth had more limited support for testing and classes, which would have made doing it in a different style much more difficult.

While the primary purpose of the lab was to give students a chance to experiment with synthesis, we did solicit feedback on the tool and the experience of using it vs. programming “by hand”. Of the 51 students, 82% reported that the tool was usable (either difficult, easy, or fun!), and 69% said that if they had such a tool in their language/testing library of choice, they would use it (sometimes or a lot) rather than programming by hand.

5 Implementation

The prototype tool SYNTH that comes as an artifact along with this paper is implemented as a Python library that operates by synthesizing code at module load time. Synthesized code is cached to speed up subsequent loads. The implementation is intentionally simple, trading some verbosity against complexity given its purpose as a prototype.

An example invocation is the following:

```python
@synth(num_tries = 2)
class protocol(object):
  """
  Purpose: A class that can encode and decode characters.

  >>> p = protocol()
  >>> p.encode("a")
  97
  >>> p.encode("b")
  98
  >>> p.decode(122)
  'z'
  """

  def __init__():
    pass
  def encode(str):
    pass
  def decode(num):
    pass


@specifies("protocol")
@given(st.characters(codec = "latin-1"))
def test_roundtrip(str):
  p = protocol()
  encoded = p.encode(str)
  decoded = p.decode(encoded)
  assert decoded == str, f"roundtrip output: {decoded} does not equal original string {str}"

synth_start(globals())
```

5.1 Synthesis

The implementation works via two stages: first, decorators @synth and @specifies record information into globals. Second, synth_start initiates synthesis/validation. A module may have any number of units of synthesis, but is intended to have only a single invocation of synth_start.

The former, @synth, indicates that the given class or function should be synthesized. As an optional argument, it takes num_tries, which changes the number of synthesis attempts it will make from the default (a parameter of the library, currently three). For a function, the existing implementation will be ignored, so by convention we put **pass** as a placeholder, though any body can exist (as it will be replaced). When @synth is put in front of a class, as above, it indicates that any methods whose bodies are **pass** should be synthesized; methods which have non-**pass** implementations should be left alone.

While the synthesis will use any doctest tests[8] for functions, classes, or methods, for validation (see §5.2), we also have support for writing property-based tests using the library Hypothesis [14]. These are declared using the @specifies decorator, which gives the name of the function or class whose synthesis should be contingent on this property passing. These will be recorded and then used for validation.

Finally, at the bottom of the file, there is an invocation synth_start(globals()) that starts synthesis. This goes through every class or function that has been annotated with @synth, first checks if the definition has already been synthesized by constructing a hash key that includes all of the docstrings, tests, and any dependencies (see §5.3). If it has, that implementation is used and the identifier is updated in the environment to the already synthesized implementation. If there is no existing implementation, then a query is sent to the LLM that includes the docstrings (which may have doctest tests), any property based tests relevant to that synthesis. For the above example, the query sent is as follows:

````
system
You are an assistant who completes Python functions or classes given names,
docstrings, and assertions that they must pass. You must pay attention to
any doctests given in the docstring, and also to any property-based tests given
separately.

user

```python
@synth(num_tries = 2)
class protocol(object):
  """
  Purpose: A class that can encode and decode characters.
````

[8] These are tests written as interactions with the REPL, e.g., in the examples above, >>> p.encode("a")
````

````
  >>> p = protocol()
  >>> p.encode("a")
  97
  >>> p.encode("b")
  98
  >>> p.decode(122)
  'z'
  """

  def __init__():
    pass
  def encode(str):
    pass
  def decode(num):
    pass
:
```
Note: please only include code for the definition to be completed; do not explain
it or include functions and/or classes we said you know about.

 You are given these property-based specifications:

```python
@specifies("protocol")
@given(st.text(alphabet=string.ascii_letters, max_size=1, min_size=1))
def test_roundtrip(str):
  p = protocol()
  encoded = p.encode(str)
  decoded = p.decode(encoded)
  assert decoded == str, f"roundtrip output: {decoded} does not equal original string {str}"
```
````

The model then responds with some text, and we attempt to extract out code from it via simple heuristics based on the use of markdown code blocks to delineate code. In this case the model responds with a valid implementation:

```python
class protocol(object):
  def __init__(self):
    pass

  def encode(self, str):
    return ord(str)

  def decode(self, num):
    return chr(num)
```

Next, the code is evaluated in a special environment containing only declared dependencies (see §5.3). This is a lightweight form of security, suitable for the attacker model: the LLM might generate code that is accidentally malicious, but not code that uses sophisticated environment escaping tricks before executing malicious code. If that produces an error, due to, e.g., the code trying to use a library that was not
```

declared (see §5.3), the error is added as a new message and the conversation is sent back to the model for another response. Thus, the model has the original prompt, its erroneous code, the error it produced, and is then asked to produce a new response.

5.2 Validation

Once the code evaluates, we validate it against the specification provided by the programmer. This is both in the form of the doctest tests (in the docstrings) and property based tests (using @specifies). While both are provided to the model for the prompt, the model is completing text; it does not run code and thus may produce code that does not satisfy the tests. If either of these fail to run, just as with code that throws errors when evaluated, we add a message with the test failure and ask the model for another response.

As an example: while the above serialization is quite easy for the synthesis to figure out, since it simply uses the ASCII numeric code for the letters, we can make it more challenging by changing our unit tests to specify "a" should convert to 0, "A" to 1, "b" to 2, etc – i.e., lowercase to even numbers, uppercase to odd. For our property-based tests, we'll leave the same round-trip property in place. In this case, the current model we are testing with (gpt-4-turbo) struggles with the task. It often generated complex code that seems to be attempting to accomplish the task, yet fails. e.g., one very close response was:

```
class protocol(object):
  def __init__(self):
    pass

  def encode(self, char):
    if char.islower():
      return ord(char) - ord('a') * 2
    else:
      return (ord(char) - ord('A')) * 2 + 1

  def decode(self, num):
    if num % 2 == 0:
      return chr((num // 2) + ord('a'))
    else:
      return chr((num // 2) + ord('A'))
```

The bug here is subtle – a missing set of parentheses in the char.islower() branch of encode, which causes expected test failures. The response SYNTH sends to the model is:

```
The code you generated failed one or more of the test cases we gave you as an assertion:
**********************************************************************
File "builtins", line ?, in protocol
Failed example:
    p.encode("a")
Expected:
```

```
    0
Got:
    -97
**********************************************************************
File "builtins", line ?, in protocol
Failed example:
    p.encode("b")
Expected:
    2
Got:
    -96

 Can you try again?
```

Unfortunately, in the particular run that generated that code, the model did not identify the missing parentheses, but rather attempted a different (also wrong) attempt. The actual results, of course, depend on particular models, which are changing all the time, and are non-deterministic; the important part is that SYNTH (or any tool like it) *validates* any code that is generated by the model. As the models get better, it is more likely that this validation will succeed (whether initially or on retry). What we do not (and should not) require is that the programmer read the code and try to figure out why the code was wrong. Instead, they should either add tests to make their specification more detailed so the model can better deal with it, or break the problem into smaller parts if the problem is too complex.

Assuming we do get a response within the bounded number of retries that satisifies all of the tests, we update the environment with this code, cache the response, and move on to the next item to synthesize.

5.3 Dependencies

Clearly, real code is not primarily composed of individual classes or functions that are entirely self contained. Most code either depends on external libraries or other parts of the code base. This provides an interesting challenge for synthesis, as the size of the input to the model is limited, so we can't simply provide the entire code base and all available libraries as input.

While for well known open source libraries (or modules from the standard library), the models might already know about them, in larger code bases the amount of code the model doesn't know about will get larger. One approach is to try to fine-tune the model on the code base, which may help, but imprecision will always cause problems, as to some extent this is more of a search problem than an AI problem.

We take a simpler approach: dependencies must be declared explicitly. They may not all be necessary, but any functions, classes, or modules that a synthesized class or function depends upon must be explicitly declared. This is done via another decorator, `@synthdepend`. This seems like a reasonable compromise: the low-level details of how an implementation is constructed is still deferred to the model, but higher level architectural decisions, including which components are built using which other components, are determined (at least approximately) by the programmer.

If we reflect, this also matches how a programmer would work. Except in very small systems, they wouldn't know everything that was available, but would rather start by researching what existing code might be useful in accomplishing the task. Once they figured that out, they would move onto implementation. Here, they still do the first step, but rather than working on the implementation, they just record the functions / classes that they think may be useful for the implementation.

Consider the following example:

```
@dataclass
class Money:
    dollars : int
    cents : int


@synth()
@synthdepend(Money)
def get_in_cents(money : Money) -> int:
    """ converts money into a total number of cents

    >>> get_in_cents(Money(1,0))
    100

    >>> get_in_cents(Money(5,5))
    505

    """
    pass


synth_start(globals())
```

The way that @synthdepend works is two-fold. First, it modifies the query that is sent to the LLM. For functions or classes that are defined in the same module, it provides the full definitions. For the above, the query that is sent looks like:

````
You know about:
```python
@dataclass
class Money:
    dollars : int
    cents : int

```
Use these to complete the following:
```python
@synth(num_tries = 2)
@synthdepend(Money)
def get_in_cents(money : Money) -> int:
    """ converts money into a total number of cents

    >>> get_in_cents(Money(1,0))
````
````

```
    100

    >>> get_in_cents(Money(5,5))
    505

    """
    pass
:
 ```
Note: please only include code for the definition to be completed;
do not explain it or include functions and/or classes we said you know about.
```

For modules, it provides the name of the module. The latter works for well-known libraries, though clearly a more sophisticated approach would be needed for large code-bases.

Then, when evaluating the resulting code, the specified identifiers are added to the environment that the code runs in. Without that change, if the synthesized code attempts to use identifiers that are not explicitly depended upon errors will result. In the above example, if we left out the @synthdepend, the following error would result (truncated for brevity):

```
Error raised trying to run code:
...
exec(code, globs)\n', '  File "<string>", line 1, in <module>\n',
"NameError: name 'Money' is not defined\n"
```

This will cause the library to attempt to re-query (as all errors do), but in this case, the fault is not in the synthesis but in what the programmer wrote. For this reason, while the library doesn't show the synthesized code, it *does* show errors that are triggered.

In that example, the synthesized code used an existing class, but there is no reason why synthesized code can't use other synthesized code. Indeed, one of the main hypotheses of this entire project is that one task that will remain the purview of programmers is the act of breaking large problems into small problems, where the latter can then be deferred to synthesis.

```
@synth()
def nextLetter(x):
    """
    Purpose: Given a letter, returns the letter after it in the alphabet

    >>> nextLetter('a')
    'b'
    >>> nextLetter('b')
    'c'
    >>> nextLetter('z')
    'a'
    >>> nextLetter('A')
    'B'
```
```

```
    >>> nextLetter('B')
    'C'
    >>> nextLetter('Z')
    'A'

    """
    pass
@synth()
@synthdepend(nextLetter)
def twoLettersAfter(x):
    """
    Purpose: Given a letter, returns the letter two letters after it in the alphabet

    >>> twoLettersAfter('a')
    'c'
    >>> twoLettersAfter('b')
    'd'
    >>> twoLettersAfter('z')
    'b'
    >>> twoLettersAfter('A')
    'C'
    >>> twoLettersAfter('B')
    'D'
    >>> twoLettersAfter('Y')
    'A'
    >>> twoLettersAfter('Z')
    'B'
    """
    pass

synth_start(globals())
```

This is a straightforward task, and synthesis succeeds. One implementation we got for twoLettersAfter is the following, showing that it used the helper function that it synthesized first.

```
def twoLettersAfter(x):
    return nextLetter(nextLetter(x))
```

If we change the definition of nextLetter to not wrap around at the end of the alphabet, it is no longer useful for twoLettersAfter. The primary model we have tested with (gpt-4-turbo) persistently attempts to use nextLetter when it is provided with @synthdepend, though switching to the model gpt-4 correctly responds to the test failures from its initial attempt to use the helper:

```
I apologize for the confusion. There seems to be a misunderstanding about how
`nextLetter` function is working, particularly for the case when we reach 'Z'
or 'z'.

Here is a correct implementation of `twoLettersAfter`:
...
```

The model gpt-4o (released shortly before submission) also correctly handles the initial erroneous attempt to use the helper, and responds better to our prompts by not even giving a description, just the corrected code.

This highlights an important aspect of this work: the concrete performance will continue to improve, and what is important to focus on is the framework within which we can harness these tools, as it is unlikely that their nature will fundamentally change. Indeed, this research is not about the models themselves (which, in the versions we use, are proprietary, opaque, can change from day to day or week to week), but rather systems that can be built using any available models. The difference, of course, is that the better the model performs, the more likely a conformant implementation can be found given a sufficiently expressive specification.

6 Related & Future Work

There is extensive work on adapting LLMs to the work of non-expert programmers, and challenges therein. Some of them are described by [23], including issues of code correctness, failure to decompose the problem down far enough, etc. They identify these issues, and more, as they note that using an LLM is a different activity that requires a different expertise that many programmers, especially non-experts, will not have or necessarily want to learn. Interestingly, using LLMs successfully requires more software engineering discipline than traditional programming, because they only work well when given small, well specified tasks. This may, ironically, be why experts find them intriguing, but does not bode well for their wide distribution. Similar work in this vein, exploring how real programmers use these tools includes [3].

Less work exists in trying to provide the power of LLMs without any direct access to prompts or the resulting code: while a few open source projects have experimented with this,[9] they've done it without any concern for validation, and as a result, there is no indication that they have produced anything that programmers could rely upon.

The broader question of validation of LLM output has been studied, and indeed, identified as an important thing that programmers struggle with [5, 13, 18, 25]. One effort to address this via a tool is a study by [10], where they provided a live programming environment to execute synthesized code. This seemed to help, though with the obvious limitation that tasks have to be relatively self-contained for the live environment to work.

We propose a different solution, in that programmers should be able to benefit from LLMs without directly interacting with either the input (the prompt) or the output (the code to review). There is certainly, as yet, much future work.

Concretely, there are questions of how to handle more extensive dependencies. For example, in a large project with thousands of API functions, having to list every one

[9] e.g., [2], which did no validation of functional correctness, and is abandoned, or [22], which uses LLMs directly to compute outputs, rather than generating code, and again, has no validation.

that is relevant to a particular new function does not seem viable. Clearly, if functions are small, they would only use a small surface of the API, but perhaps identifying which functions are most relevant could also be deferred to the LLM.

One possibility we have thought of is constructing sets of functions, or essentially labeled portions of an API, and declaring that the newly synthesized function may need to use that. As an example, a labeled set may be the functions that are used for caching. Likely, rather than providing the full definitions of all of these functions, their signatures and documentation would be sufficient for the synthesis.

Another concrete, and easier, task would be to support more mechanisms of testing. Not only, within the context of Python, the built-in `unittest` library, but likely libraries that support more complex setup, teardown, and mocking for tests.

This would likely help with another related challenge: handling global mutable state. Testing code that involves global mutation is challenging, and SYNTH inherits this challenge, since it must test code in order to validate it. This is a well-known and well-studied problem though, and tools like SYNTH can directly benefit from significant software engineering work on this problem.

More challenging and more abstract challenges exist when envisioning how to turn the prototype outlined in this paper and realized by the tool SYNTH into a real system that programmers could use. Clearly, it would need to have wide support for testing frameworks and better dependency support. But, it may also need to have more fine tuning and debugging capability. While, in principle, we are opposed to programmers inspecting the output of the synthesis, we acknowledge that its inevitably going to be something that programmers want to do. It's also possible that they would want to add custom prompt injections, beyond the standard ones generated, though obviously the docstrings could have more explicit prompt instructions as well.

7 Conclusion

Large-Language Models have promised to revolutionize, eliminate, or at least, fundamentally alter programming. And yet, all existing tools rely fundamentally on the flawed premise that programmers will validate the output to rule out mistakes, security vulnerabilities, or other bugs that the generated code may have.

In this paper, we have presented an actual vision for the future, where AI-driven programming does not rely upon a programmer-as-oracle, carefully curating the output of the model to ensure it only produces sensible results. Rather than serving as a copilot, in dialog with the programmer, we envision AI-based tools as powering non-deterministic compilers that take specifications (in the form of documentation, tests, etc) and transform them into code. Critically, that code must be validated against the programmers expectations, expressed as executable specifications. By viewing synthesis as a compilation problem, the normal task of compiler correctness leads to this novel solution. Just as a compiler should preserve the semantics of a program, synthesis should preserve the semantics of the specification.

While we have focused, in this paper, on specifications, in the form of unit and property based tests, that relate to functional correctness, any specification that can be validated automatically would fit naturally into this framework. For example, we could easily envision performance requirements expressed as benchmarks that are then used to validate synthesized code against certain minimum space or time requirements. The key is to escape the trap of autocompletion, which arises naturally from the text-based nature of the models, but is at odds with the semantic content of programs, which is about behavior, not about written text.

Indeed, once we view LLMs as tools that should be used but not seen by programmers, many different ideas appear; e.g., given a set of example data and a partial specification for an implementation, we could synthesize an implementation and then run it on the set of examples, providing the resulting outputs as proposed test cases for the programmer to inspect, helping them to construct a more thorough specification by looking at concrete cases that they have identified as important.

In the end, we do believe that LLMs will fundamentally alter programming, but that they will do it more along the lines of prior advances in compilers and language runtime systems rather than by any "end of programming". The ability of even the current models to generate code is already quite impressive, but validation is, indeed, the key challenge, and one that cannot be dismissed lightly.

References

[1] Stability AI. *StableCode*. Accessed May 28, 2024. URL: https://stability.ai/news/stablecode-llm-generative-ai-coding.

[2] Banana. *Fructose: LLM calls as strongly-typed functions*. Accessed May 28, 2024. URL: https://github.com/bananaml/fructose.

[3] Shraddha Barke, Michael B. James, and Nadia Polikarpova. "Grounded Copilot: How Programmers Interact with Code-Generating Models". In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (Apr. 2023). DOI: 10.1145/3586030.

[4] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. "Modern code reviews in open-source projects: which problems do they fix?" In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: Association for Computing Machinery, 2014, pages 202–211. ISBN: 9781450328630. DOI: 10.1145/2597073.2597082.

[5] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. "Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools". In: *Queue* 20.6 (Jan. 2023), pages 35–57. ISSN: 1542-7730. DOI: 10.1145/3582083.

[6] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. "Code reviews do not find bugs: how the current code review best practice slows us down". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. ICSE '15. Florence, Italy: IEEE Press, 2015, pages 27–28. DOI: 10.1109/ICSE.2015.131.

[7] Yangruibo Ding, Marcus J. Min, Gail Kaiser, and Baishakhi Ray. "CYCLE: Learning to Self-Refine the Code Generation". In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (Apr. 2024). DOI: 10.1145/3649825.

[8] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner. "An Empirical Study on the Effectiveness of Security Code Review". In: *Engineering Secure Software and Systems*. Edited by Jan Jürjens, Benjamin Livshits, and Riccardo Scandariato. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pages 197–212. ISBN: 978-3-642-36563-8. DOI: 10.1007/978-3-642-36563-8_14.

[9] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. "The Structure and Interpretation of the Computer Science Curriculum". In: *J. Funct. Program.* 14.4 (July 2004), pages 365–378. ISSN: 0956-7968. DOI: 10.1017/S0956796804005076.

[10] Kasra Ferdowsi, Ruanqianqian (Lisa) Huang, Michael B. James, Nadia Polikarpova, and Sorin Lerner. "Validating AI-Generated Code with Live Programming". In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. CHI '24. <conf-loc>, <city>Honolulu</city>, <state>HI</state>, <country>USA</country>, </conf-loc>: Association for Computing Machinery, 2024. ISBN: 9798400703300. DOI: 10.1145/3613904.3642495.

[11] Hugingface. *SafeCoder*. Accessed May 28, 2024. URL: https://huggingface.co/blog/safecoder.

[12] Anjali Khurana, Hariharan Subramonyam, and Parmit K Chilana. "Why and When LLM-Based Assistants Can Go Wrong: Investigating the Effectiveness of Prompt-Based Interactions for Software Help-Seeking". In: *Proceedings of the 29th International Conference on Intelligent User Interfaces*. IUI '24. <conf-loc>, <city>Greenville</city>, <state>SC</state>, <country>USA</country>, </conf-loc>: Association for Computing Machinery, 2024, pages 288–303. ISBN: 9798400705083. DOI: 10.1145/3640543.3645200.

[13] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. "A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. <conf-loc>, <city>Lisbon</city>, <country>Portugal</country>, </conf-loc>: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3608128.

[14] David R. MacIver. *Hypothesis*. Accessed May 29, 2024. URL: https://hypothesis.readthedocs.io/en/latest/.

[15] Meta. *Code Llama*. Accessed May 28, 2024. URL: https://ai.meta.com/blog/code-llama-large-language-model-coding/.

[16] Microsoft. *Github Copilot*. Accessed May 28, 2024. URL: https://github.com/features/copilot.

[17] Roberto Minelli, Andrea Mocci, and Michele Lanza. “I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time”. In: *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. ICPC ’15. USA: IEEE Computer Society, 2015, pages 25–35. ISBN: 9781467381598. DOI: 10.1109/ICPC.2015.12.

[18] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. “Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming”. In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. CHI ’24. <conf-loc>, <city>Honolulu</city>, <state>HI</state>, <country>USA</country>, </conf-loc>: Association for Computing Machinery, 2024. ISBN: 9798400703300. DOI: 10.1145/3613904.3641936.

[19] OpenAI. *ChatGPT*. Accessed May 28, 2024. URL: https://openai.com/index/chatgpt/.

[20] David Lorge Parnas. “Software aspects of strategic defense systems”. In: *Commun. ACM* 28.12 (Dec. 1985), pages 1326–1335. ISSN: 0001-0782. DOI: 10.1145/214956.214961.

[21] Rajshakhar Paul, Asif Kamal Turzo, and Amiangshu Bosu. “Why Security Defects Go Unnoticed during Code Reviews? A Case-Control Study of the Chromium OS Project”. In: *Proceedings of the 43rd International Conference on Software Engineering*. ICSE ’21. Madrid, Spain: IEEE Press, 2021, pages 1373–1385. ISBN: 9781450390859. DOI: 10.1109/ICSE43902.2021.00124.

[22] Prefect. *Marvin: The AI engineering toolkit*. Accessed May 28, 2024. URL: https://github.com/prefecthq/marvin.

[23] Advait Sarkar, Andrew D. Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. *What is it like to program with artificial intelligence?* 2022. arXiv: 2208.06213 [cs.HC].

[24] Linda J Skitka, Kathleen Mosier, and Mark D Burdick. “Accountability and automation bias”. In: *International Journal of Human-Computer Studies* 52.4 (2000), pages 701–717. DOI: 10.1006/ijhc.1999.0349.

[25] Helena Vasconcelos, Matthew Jörke, Madeleine Grunde-McLaughlin, Tobias Gerstenberg, Michael S. Bernstein, and Ranjay Krishna. “Explanations Can Reduce Overreliance on AI Systems During Decision-Making”. In: *Proc. ACM Hum.-Comput. Interact.* 7.CSCW1 (Apr. 2023). DOI: 10.1145/3579605.

[26] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. 2021. arXiv: 2109.00859 [cs.CL].

[27] Matt Welsh. “The End of Programming”. In: *Commun. ACM* 66.1 (Dec. 2022), pages 34–35. ISSN: 0001-0782. DOI: 10.1145/3570220.

[28] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. “Measuring Program Comprehension: A Large-Scale Field Study with Professionals”. In: *IEEE Trans. Softw. Eng.* 44.10 (Oct. 2018), pages 951–976. ISSN: 0098-5589. DOI: 10.1109/TSE.2017.2734091.

About the authors

Daniel Patterson dbp@dbpmail.net
https://orcid.org/0000-0002-2116-8684

Smaran Teja smaranteja32@gmail.com
https://orcid.org/0009-0009-2854-7554

Lukas Tegge lukasgrat@gmail.com
https://orcid.org/0009-0002-1594-3549

Rachel Chen chen.rachel@northeastern.edu
https://orcid.org/0009-0003-6477-649X