What do LLMs need to Synthesize Correct Router Configurations?

Rajdeep Mondal
UCLA
USA
mondalrajdeep14@ucla.edu

Alan Tang
UCLA
USA
atang42@cs.ucla.edu

Ryan Beckett
Microsoft Research
USA
Ryan.Beckett@Microsoft.com

Todd Millstein
UCLA
USA
todd@cs.ucla.edu

George Varghese
UCLA
USA
varghese@cs.ucla.edu

Abstract

We investigate whether Large Language Models (e.g., GPT-4) can synthesize correct router configurations with reduced manual effort. We find GPT-4 works very badly by itself, producing promising draft configurations but with egregious errors in topology, syntax, and semantics. Our strategy, that we call *Verified Prompt Programming*, is to combine GPT-4 with verifiers, and use localized feedback from the verifier to automatically correct errors. Verification requires a specification and actionable localized feedback to be effective. We show results for two use cases: translating from Cisco to Juniper configurations on a single router, and implementing a no-transit policy on multiple routers. While human input is still required, if we define the *leverage* as the number of automated prompts to the number of human prompts, our experiments show a leverage of 10X for Juniper translation, and 6X for implementing the no-transit policy, ending with verified configurations.

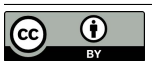
CCS Concepts

• **Software and its engineering** → **Application specific development environments**.

Keywords

CoSynth, network verification and synthesis, large language models (LLMs)

ACM Reference Format:
Rajdeep Mondal, Alan Tang, Ryan Beckett, Todd Millstein, and George Varghese. 2023. What do LLMs need to Synthesize Correct Router Configurations?. In *The 22nd ACM Workshop on Hot Topics in Networks (HotNets '23), November 28–29, 2023, Cambridge, MA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626111.3628194



HotNets '23, November 28–29, 2023, Cambridge, MA, USA

ACM ISBN 979-8-4007-0415-4/23/11.
https://doi.org/10.1145/3626111.3628194

1 Introduction

While GPT-4 and other large language models (LLMs) have shown great success in some domains (e.g., writing poems, passing the LSAT) they have been shown to have issues in other domains (e.g,. math, word puzzles) [3]. Language models have had some success in helping users write sequential programs in systems like AlphaCode [10], CoPilot [7], Codex [4] and Jigsaw [8]. They have also been explored as promising assistants for software testing and debugging [13]. Our work investigates code generation by LLMs for a different domain. We examine GPT-4's ability to write router configuration files, traditionally written by humans, that help tune routes and forwarding decisions and are critical for network operation. Our early experiments show that GPT-4 by itself is an "idiot-savant", capable of brilliance but also making simple errors that an operator would be fired for making.

Critics have derided LLMs as mere "stochastic parrots" [2], because they produce text (say of a program) *syntactically* by predicting the next word based on a statistical model derived by training on a vast corpus of text from the Internet. Our broader goal beyond synthesizing configs is to see whether LLMs can be fused with other *programs* (via APIs) to resemble a "stochastic owl" that understands program semantics.

A plausible way to introduce semantics is to pair a LLM with an automatic verifier such as a SAT solver or a model checker. But verification is not a panacea. First, a verifier cannot prove correctness without a specification. In practice, specifications are incomplete, so not all solutions are in fact acceptable to the user. Second, for the verifier to automatically (with minimal human aid) interact with the LLM, the verifier must provide actionable feedback. We found it was easier for the LLM to correct itself using feedback from *modular verification* of components of a network (individual routers [11] or even route maps within a router [12]), rather than the network as a whole.

Figure 1 shows the traditional method of *pair programming (PP)*, embodied in systems like GitHub CoPilot [7], where a

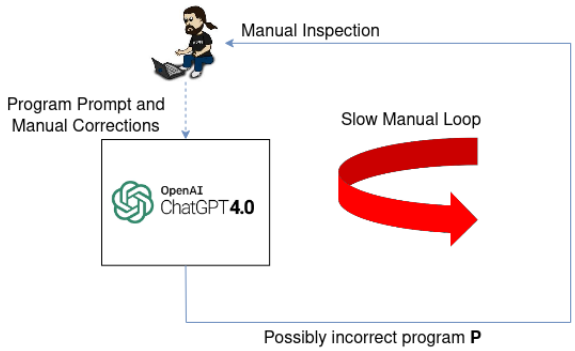


Figure 1: Pair Programming using human correction

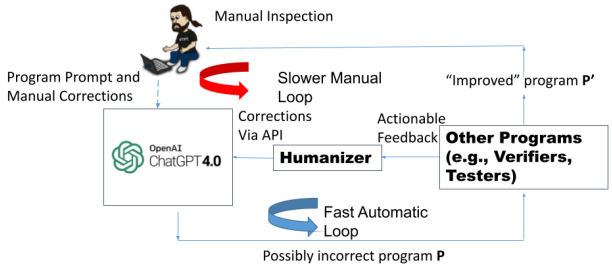


Figure 2: Verified prompt programming

human and an AI work together to author a program. In pair programming, the AI and the human form a tuple (A, H) and the human H *manually* checks for correctness of the output of the AI A and then *manually* issues correction prompts to A as shown in the figure. Such manual initial prompting and subsequent manual correction is often called *prompt engineering*.

Figure 2 shows our alternate vision. In what we call *Verified Prompt Programming (VPP)*, the AI, the human, and a verification suite (V) form a triple (A, H, V). The verification suite checks for correctness and automatically issues localized corrections. V may abandon automatic correction after some number of trials, and the human must still correct manually. However, our hypothesis is that human effort is reduced as the output grows "closer" to a correct program.

Notice that there is a fast inner loop between V and A, where verifier results are automatically fed back to GPT-4. Since verifier feedback is often cryptic, we use simple code that we call a *humanizer* that converts the feedback to natural language prompts that are given to GPT-4. When V either determines the final configuration is correct or a time bound elapses, V sends the output back to the user as part of the slow manual loop. We examine a "reduced work hypothesis": that the work in the manual loop in Figure 2 is significantly less than then the manual work in Figure 1

To quantify reduced human effort we introduce a simple measure that may be useful in other VPP contexts. Define *leverage* as the ratio L of the number of automated prompts in Figure 2 to the number of human prompts. Leverage measures the effect of the *verifier suite*, the potential improvement in going from (A, H) to (A, V, H), keeping the language model A and the human H the same. Note that the leverage can differ across multiple iterations of the same experiment, due to the stochastic nature of the LLM output.

The reader may think the real leverage is the improvement from H to (A, H), or from H to (A, V, H). But this depends on the capability of the human H and is hard to make uniform or repeatable. Given how error-prone (A, H) is for configurations, we find it more natural to measure the improvement caused by VPP. Our definition also assumes every automatic correction in Figure 2 would otherwise be done by a human in Figure 1.

The reduced work hypothesis is that the leverage $L > 1$ is high. Even if the leverage is low (say 1), since it is crucial that router configurations be correct, combining with a verifier seems critical. We were happy to find that in both use cases we did end with verified configurations via GPT-4: this was not obvious at the outset.

This vision and hypothesis extends beyond synthesizing configs to more general programs. Prompt programming (as opposed to prompt engineering) also reflects the use of APIs and automatically generated feedback prompts that may be more generally useful. However, network configs are a simple enough domain to experiment with. Further, there exist config verifiers (e.g., Campion [12] and Lightyear [11]) that provide actionable localized feedback.

For the rest of this paper, we examine the reduced manual work hypothesis and measure leverage for two use cases: translating a config on a single router from Cisco to Juniper syntax, and implementing a simple policy ("no transit") on a network of 6 routers. We conducted these experiments during February-March 2023. Section 2 describes the system organization of a potential system we call CoSynth. Section 3 describes experiments with Cisco to Juniper translation, while Section 4 describes implementing no-transit on multiple routers. Section 5 compares our ideas to previous work and Section 6 describes lessons learned.

2 System Organization

Figure 3 is a refinement of the more general Verified Pair Programming (VPP) vision of Figure 2 that we call CoSynth. We emphasize we have not built CoSynth. While we use GPT-4 we have not been able to access the APIs, and so manually simulated the API calls with prompts to ChatGPT. Our goal is not to demonstrate a working system but instead to explore GPT-4's ability to author configurations, as in the "Sparks of AGI" paper [3].

The verification suite shown in Figure 3 consists minimally of two verifiers, a syntax verifier (we used Batfish [6]) and a semantics verifier (we used different ones depending on the use case). For our second use case, we used a third verifier, a topology verifier (that we wrote in Python) as we found that GPT-4 sometimes missed announcing routes to neighbors. The user provides a precise natural language description of the context (topology, routers, interfaces) and the desired task (e.g. the Cisco config and a request to translate it to Juniper).

GPT-4 output is fed first to Batfish to check for syntax errors. CoSynth sends GPT-4 feedback about erroneous lines, "humanized" in natural language (see Table 1 for examples). The boxes labelled **H** in Figure 3 correspond to the humanizer in Figure 2, which acts as an error parser and natural language translator.

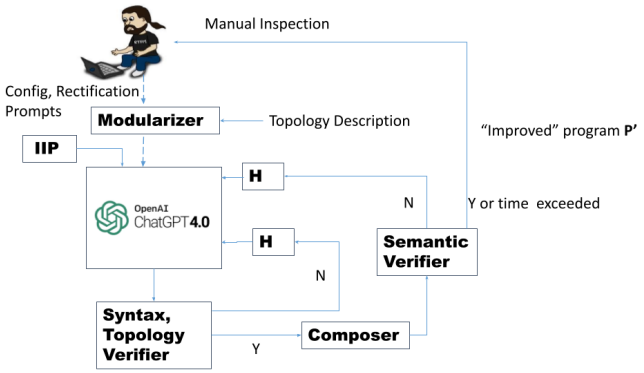


Figure 3: Verified prompt programming for Configs

If all syntax errors are corrected (if too many syntax correction attempts occur, CoSynth punts to the user), the output is passed to the semantics verifier. For our first use case, we use Campion [12] as a verifier. For our second use case we use Batfish's symbolic route map analysis as the verifier, asking it to verify local policies that together ensure the desired global policy, as in Lightyear [11]. Once again, the semantic verifier feedback is passed back, suitably humanized, to GPT-4. We found that GPT-4 would sometimes correct a semantic error while introducing a new syntax error, in which case we had to return to the syntax verifier. When the semantic verifier attests to a correct config or too many correction attempts transpire, CoSynth returns to the human.

When CoSynth works with multiple routers, we used another module called a "Modularizer" (Figure 3). For network configs, the idea is that we start with a precise machine readable (we use JSON) description of the "modules" which in our case is the topology and the connections. The Modularizer outputs a sequence of Natural Language Prompts that describes the topology to GPT-4 (e.g.,. Router $R1$ is connected to Router $R2$ via interface $I1$ at $R1$ and $I2$ at $R2$). The Composer puts back the pieces (in our case in a folder for Batfish).

The modularizer follows the prompt engineering paradigm "Give the Model Time to Think" [5], which suggests breaking a complex prompt into simpler sub-prompts. Exploiting modularity is a way to do so for program synthesis. A second technique we find useful is what is called single shot prompting [5]. We start each chat with a set of *initial instruction prompts (IIP)* (Figure 3) loaded from a database for avoiding common mistakes. The IIP database can be built and added by experts over time. The I/O examples in Jigsaw [8] are an IIP, but our IIP contains instructions rather than examples.

3 Cisco to Juniper Translation

We translate a Cisco configuration into an equivalent Juniper one using Verified Prompt Programming. Batfish [6] is used to identify syntax errors. Campion [12] is used to detect and localize semantic differences that are used to refine the result. We show examples of the issues encountered, and discuss success and limitations of the approach.

3.1 Method

First, we provide the Cisco configuration, and the prompt: "Translate the configuration into an equivalent Juniper configuration." GPT-4 will produce a translation into Junos format that typically contains several errors and differences. We then try to rectify these errors iteratively, using "humanized" feedback from the verifiers. We re-verify the entire configuration on each iteration. For our experiment we focus exclusively on behavior related to routing and forwarding, ignoring potentially important features such as NTP servers.

To design the humanizer, which is a Python script, we distinguish four classes of configuration errors:

Syntax errors: Batfish produces parse warnings identifying relevant lines that do not use valid Juniper syntax.

Structural mismatch/conflict: This is when a component, connection, or named policy is present in the original configuration but not in the translation (or is present in the translation but not the original). For example, if the original configuration defined a BGP neighbor but there is no corresponding neighbor in the translation, there would be a mismatch in the routing connections. Campion is able to detect this, and identify the missing or extra items.

Attribute differences: This is when a numerical attribute has a different value between the two configurations. An example is OSPF link cost difference between two corresponding interfaces. Campion detects these and prints the attributes for corresponding components.

Policy behavior differences: This is when a route map or access control list has a semantic difference. Route maps are used to filter incoming or outgoing route advertisements, so a difference would mean that that there are some route advertisements that are allowed by one router but not allowed by the other. Campion is able to detect these and output the relevant policy names, prefixes, and lines for these differences.

The distinction among errors helps for two reasons. First, syntax errors and structural mismatches have to be handled earlier since they can mask attribute differences and policy behavior differences. Second, different types of errors require different humanized prompts, while errors of the same type can reuse similar prompts. Each type of error can be summarized with a formulaic prompt with some fields inserted based on the error reported by Batfish or Campion.

Table 1 shows the formulas and examples of generated prompts. Batfish parse errors and warnings can be reused as prompts for syntax errors. Prompts for structural mismatches and attribute differences are easily generated from the relevant components and attributes. Policy behavior differences are

Type	Generated Prompt
Syntax error	There is a syntax error: '*policy-options prefix-list our-networks 1.2.3.0/24-32*'
Structural mismatch	In the original configuration, there is *an import route map for bgp neighbor 2.3.4.5*, but in the translation, there is no corresponding *route map*
Attribute difference	In the original configuration, *the OSPF link for Loopback0* has *cost* set to *1*, but in the translation, the corresponding *link to lo0.0* has *cost* set to *0*
Policy behavior difference	In the original configuration, for *the prefix 1.2.3.0/25*, the *BGP export policy to_provider for BGP neighbor 2.3.4.5* performs the following action: *ACCEPT*. But, in the translation, the corresponding *BGP export policy to_provider* performs the following action: *REJECT*

Table 1: Sample rectification prompts for translation generated using formulas (non-italicized text), and fields generated from Batfish and Campion (italicized text).

more difficult since it is not always clear how to describe the affected input space that is treated differently. We opt for the approach of giving a single concrete example.

3.2 Experience and Results

Error	Type	Fixed
Missing BGP local-as attribute	Syntax error	Yes
Invalid syntax for prefix lists	Syntax error	Yes
Missing/extra BGP route policy	Structure conflict	Yes
Different OSPF link cost	Attribute error	Yes
Different OSPF passive interface	Attribute error	Yes
Setting wrong BGP MED value	Policy error	Yes
Different prefix lengths match in BGP	Policy error	No
Different redistribution into BGP	Policy error	No

Table 2: Translation errors found and whether GPT-4 was able to fix them with generated prompts.

We tried translating a Cisco configuration from the Batfish examples [6] into Juniper format. This configuration was short enough to fit within GPT-4 text input limits, but used non-trivial features including BGP, OSPF, prefix lists, and route maps. Progress is not monotonic: GPT-4 can fix one error but introduce new errors that were not previously there. Sometimes it even reintroduces errors that were previously fixed! However, we were ultimately able to succeed in the translation task, with a mix of automated and manual prompts.

Leverage: In one such test run, the entire cycle of prompts was 2 human prompts and 20 automated prompts, for a leverage of 10X. Some of the 20 automatic prompt correction cycles included minor cycles for syntax correction not just at the start but also after correcting semantic errors. To be clear, we "simulated" each API call by feeding our automatically generated prompts manually to GPT-4.

Table 2 shows errors in the translation at some point and whether GPT-4 was able to fix them using an automatically generated prompt. In more detail:

Missing BGP local-as attribute: The translated BGP neighbor declarations did not include a local AS attribute. We label this a syntax error since it produces a parse warning.

Missing/extra BGP routing policy: An import or export policy is used for a BGP neighbor in only one configuration.

Different OSPF link attributes: OSPF links have a number of attributes, and the translation sometimes contains differences in link cost or passive interface settings.

Setting wrong BGP MED value: The translation of one BGP routing policy did not update the BGP MED value. This was caused by an error in translating one of the route map clauses from the original Cisco configuration.

Different Redistribution behavior into BGP: Cisco and Juniper formats handle route redistribution into BGP differently. Juniper typically does this using the same routing policies that control importing and exporting BGP routes while Cisco configurations set a separate route map for route redistribution. In our case, Campion detected that the Juniper configuration was redistributing some routes that the Cisco configuration did not. This could be fixed by adding a "from bgp" condition to a number of locations in the policy. Unlike the previously described errors, GPT-4 was unable to fix this when given the automatically generated prompt. Instead it usually does nothing when asked to fix the error. However, it was able to fix the problem when asked more directly to add "from bgp" conditions to routing policies.

BGP prefix list issues: Another subtle issue occurred when translating prefix lists. The original Cisco configuration contains the following prefix list:

ip prefix-list our-networks seq 5 permit 1.2.3.0/24 ge 24

The noteworthy part is the "ge 24" which says to match prefixes with length 24 or greater. There is no equivalent of this syntax in Juniper, but for our use case, there are at least two methods of getting similar behavior in Juniper with different syntax. When GPT-4 is asked to translate the configuration, it usually just omits the "ge 24" part, so the space of prefixes matched will differ in the translation. When asked to fix this problem, it sometimes generates configurations with incorrect syntax. For example, it can output the following:

prefix-list our-networks { 1.2.3.0/24-32; }

which is not valid Juniper syntax.

4 Global Policies via Local Synthesis

Next, we used GPT-4 to generate router configs for a given network topology based on local policies for each router, inspired by Lightyear [11], which does control plane verification by verifying local invariants. We limited our scope to BGP.

For semantic correctness, we use two new modules. The first is a "topology" verifier which checks whether the config of a particular router follows the defined topology. It checks

whether GPT-4 sets up all interfaces, declares BGP neighbors and announces networks correctly. Second, we run Batfish to check local policies defined in the prompts; the outputs are used to refine the result.

4.1 Method

We begin by specifying the task to GPT in an initial prompt using a couple of sentences. The intention is to influence the LLM to start 'thinking' in a certain fashion. Our goal is to make the network follow the no-transit policy, under which no two ISP's should be able to reach other. However, all ISPs should be able to reach the CUSTOMER and vice versa.

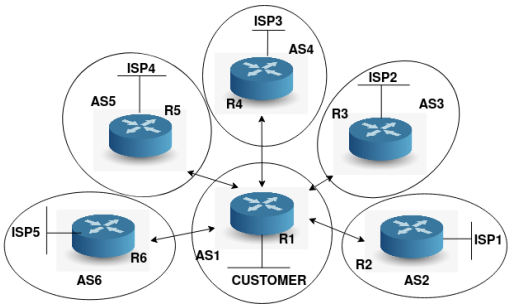


Figure 4: Star network topology used for local synthesis.

It is difficult to write a natural language description of the topology, a task prone to human error. We wrote an automated script that generates text given the topology as input. In our experiments, we limited our scope to star networks where one router would be attached to a CUSTOMER IP, while the other routers are connected to different ISPs (Figure 4). All the ISP routers are directly connected to the first router. The "network generator" therefore only needs the number of routers as input. It has two outputs: 1) a textual description and 2) a JSON dictionary for the entire network topology. The textual description is used as a prompt, while the JSON dictionary is used later to check whether the generated configs match the topology.

Local versus Global Policy Prompts? We tried specifying to GPT-4 the global no-transit policy at once. GPT-4 generated two innovative strategies: filtering routes using AS path regular expressions, and denying ISP prefixes from being advertised to other routers from the customer router. Unfortunately, we found after correcting topology and syntax errors, when we provided feedback in terms of a counterexample packet (as would be provided by a "global" network verifier like Minesweeper), GPT-4 was confused and kept oscillating between incorrect strategies. We found that specifying local policies as in Lightyear [11] gave us better results because it allowed us to localize verification errors to specific routers and specific route maps within those routers.

We asked GPT-4 to generate configs for each router using a new prompt each time, specifying the *local policy* for each router. Specifically, the policy is that $R1$ should add a specific community at the ingress to each ISP and then drop routes based on those communities at the egress to each ISP. The generated errors fell into three categories:

Syntax errors: GPT-4 generates a configuration with invalid Cisco syntax. Batfish produces parse warnings identifying these errors.

Topology errors: GPT-4 incorrectly declares or misses some BGP neighbors or forgets to announce certain networks. For this, we use an automated "topology verifier", whose main purpose is to systematically parse all the ethernet interface, BGP neighbor and network declarations within the config and match them against the network architecture listed in the JSON dictionary. It then points out all the missing declarations and topological inconsistencies.

Semantic errors / Policy errors: GPT-4 produces configs that do not follow the intended local policy. We use Batfish "Search Route Policies" for verification in this step. In case there is a semantic error, Batfish produces an example where the local policy is not followed. This example is then fed to GPT-4 in a fresh prompt.

Classifying into separate categories allowed us to use different tools to address each one. Table 3 lists examples of the rectifying prompts. Once all the errors are rectified, we simulate the entire BGP communication using Batfish as a final step, in order to ensure that the global policy is satisfied, though the proof technique of Lightyear [11] could instead be used to ensure that the local policies imply the global one.

4.2 Experience and Results

Since some GPT-4 errors were more common, we supplied it an IIP (the Inital Instruction Prompt) as follows:

CLI prompts: GPT-4 would often generate commands to enter on the Cisco command line interface, which is undesirable. Thus we specifically asked it to generate the .cfg files.

Wrong keywords: While generating the configs, it would often use certain keywords such as 'exit', 'end', 'configure terminal', 'ip routing', 'write', 'hostname' and 'conf t'. It had a tendency to place some of them in the wrong locations. Hence, we directed it not to use these keywords.

Match Community: GPT-4 sometimes tries to match directly on a community value, which is incorrect. Instead, a community list must be declared that contains the community value, and the route-map should match on the community list. Thus we included another IIP telling GPT-4 to define and match on community lists.

Adding Communities: When asked to add communities to a route using a route-map, GPT-4 generates syntax similar to:

route-map ADD_COMMUNITY permit 10
set community 100:1

The above route-map erroneously replaces all existing communities in the route with the community 100:1. So we added an initial prompt saying that it should always use the "additive" keyword when adding a community to the route.

These initial prompts along with the syntax rectification scheme of Table 3 are able to eliminate common syntax errors

Type	Examples
Syntax error	'*ip community-list standard COMM_LIST_R2_OUT permit .+*' is wrong syntax.
Topology error	1. Interface *eth0/1* ip address does not match with given config. Expected *2.0.0.1*, found *2.0.0.2* 2. Local AS number does not match. Expected *1*, found *3* 3. Neighbor with IP address *1.0.0.1* and AS *1* not declared 4. Incorrect network declaration. *7.0.0.0/24* is not directly connected to *R1*
Semantic error	The route-map *DROP_COMMUNITY* permits routes that have the community *100:1*. However, they should be denied.

Table 3: Sample rectification prompts for local synthesis. Batfish or the topology verifier provides the italicized text.

produced by GPT-4. Despite this, we found two egregious cases where human intervention is needed:

Placing neighbor commands in the wrong location: In a config file for BGP, all neighbor commands, which attach a route-map to an interface, must be placed under the "router bgp" block. Sometimes GPT-4 defines a route-map and then associates it with an interface outside the "router bgp" block. Batfish catches this syntax error, but the output is not informative enough for GPT-4 to be able to fix the issue.

AND/OR Semantics in match statements: For no-transit, we asked GPT-4 to generate a config for $R1$ that would add a specific community to every route incoming from $R2$, and similarly for the other neighbors of $R1$ (Figure 4). We also asked it to filter routes containing any such community on the egress of the interfaces connecting $R1$ to $R2 - R6$. GPT-4 added the correct communities at the ingress, but at the egress it incorrectly used AND semantics to filter routes, as in the following route-map for the $R1 - R2$ interface:

```
route-map FILTER_COMM_OUT_R2 deny 10
 match community 3
 match community 4
 match community 5
 match community 6
route-map FILTER_COMM_OUT_R2 permit 20
```

Community list 3 is associated with routes incoming from R3, community list 4 with those coming from R4, and so on. We desire routes incoming from $R3 - R6$ to be filtered out at the egress to $R2$. The above config will only filter out routes that have *all four* communities. When we asked Batfish whether the above route-map filters all routes that match the community list 3, it produced a counterexample, but this feedback to GPT-4 failed to rectify the issue. Instead, a human prompt was needed to ask GPT-4 to declare each match statement in a separate route-map stanza.

Leverage: In one such run, the entire cycle took 2 human prompts and 12 automated prompts, for a leverage of 6X.

5 Previous Work

AlphaCode [10], CoPilot [7], Codex [4] and Jigsaw [8] and numerous other recent systems use large language models for program synthesis. While they concentrate on sequential programs, the deeper difference is that they do not pair the synthesizer with verifiers. Instead, AlphaCode, Codex, and Jigsaw ask users to provide test cases and uses them to test (but not verify) the synthesized program.

Alphacode [10] does not use a general purpose LLM but instead leverages a curated data set of working programs. Codex [4] uses repeated sampling instead of correction to help generate programs that meet the test cases. Jigsaw [8] does automatic syntax correction via AST-to-AST transformations. CoPilot [7] can suggest invariants but does not attempt an axiomatic proof. These earlier systems do not address two fundamental questions that we do: how to use a specification, and how to provide localized feedback. However, their techniques are complementary to ours, and can be used to potentially improve leverage in Verified Prompt Programming.

The use of ChatGPT with the Kani Rust verifier [9] comes closest to our vision. They finesse the specification question (as we do for Cisco to Juniper) by focusing on *program transformations* for which the source program is the specification. They also do not use modularity or local specifications. More fundamentally the Kani [9] use case does not do prompt programming: the user *always* manually switches between the verifier and the LLM, precluding possible leverage.

6 Conclusions

Our experiments are very preliminary but suggest:

1. Ramanujam Effect: As with the brilliant mathematician Ramanujam, some of whose conjectures were incorrect and needed Hardy's help [1] for proofs, GPT-4 by itself is not ready for use without a verifier, making elementary errors that can bring networks down.

2. Verified Prompt Programming: Using a verifier and automated corrections via a humanizer, GPT-4 can synthesize reasonable but not completely correct configurations for simple use cases, but the leverage in reduced human effort can be high. Modular verification seems crucial to provide the LLM with actionable feedback.

3. Local versus Global Specifications: Modular synthesis is the dual to modular verification. The search space for the LLM is large, which increases the chance that it will not be able to correctly complete a synthesis task based on a global specification. Instead the user needs to decide and describe the "roles" each node plays in satisfying the global spec and provide this information to the LLM.

Much further testing in more complex use cases is needed. Can GPT-4 add a new policy incrementally without interfering with existing verified policy? While our paper is set in the context of network configuration, the vision, definitions (e.g., leverage) and lessons (e.g., the need for actionable local feedback, modularity, humanizers and IIPs) seem more generally useful to synthesize other programs.

References

[1] B. Bollobas. The man who taught infinity: how G.H. Hardy tamed Srinivasa Ramanujan's genius. https://theconversation.com/the-man-who-taught-infinity-how-gh-hardy\-tamed-srinivasa-ramanujans-genius-57585, 2023.

[2] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '21, page 610–623, New York, NY, USA, 2021. Association for Computing Machinery.

[3] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang. Sparks of artificial general intelligence: Early experiments with GPT-4, 2023.

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code, 2021.

[5] DeepLearning.AI. ChatGPT Prompt Engineering for Developers. https://learn.deeplearning.ai/chatgpt-prompt-eng/lesson/1/introduction, 2023.

[6] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. NSDI'15, page 469–483, USA, 2015. USENIX Association.

[7] github. Github CoPilot: Your AI Pair Programmer. https://github.com/features/copilot, 2023.

[8] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1219–1231, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Kani Rust Verifier Blog. Writing Code with ChatGPT? Improve it with Kani. https://model-checking.github.io/kani-verifier-blog/2023/05/01/writing-code-with-chatgpt-improve-it-with-kani.html, 2023.

[10] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, dec 2022.

[11] A. Tang, , R. Beckett, K. Jayaraman, T. Millstein, and G. Varghese. Lightyear: Using modularity to scale BGP control plane verification. SIGCOMM '23, to appear. Association for Computing Machinery, 2023.

[12] A. Tang, S. K. R. Kakarla, R. Beckett, E. Zhai, M. Brown, T. Millstein, and G. Varghese. Campion: Debugging router configuration differences. SIGCOMM '21, page 748–761, New York, NY, US, 2021. Association for Computing Machinery.

[13] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang. Software testing with large language model: Survey, landscape, and vision, 2023.