The Command Line GUIde: Graphical Interfaces from Man Pages via AI

Saketh Ram Kasibatla*, Kiran Medleri Hiremath*, Raven Rothkopf, Sorin Lerner, Haijun Xia, Brian Hempel
{skasibatla, kmedlerihiremath, rrothkopf, lerner, haijunxia, bhempel}@ucsd.edu
University of California San Diego, La Jolla, CA, USA

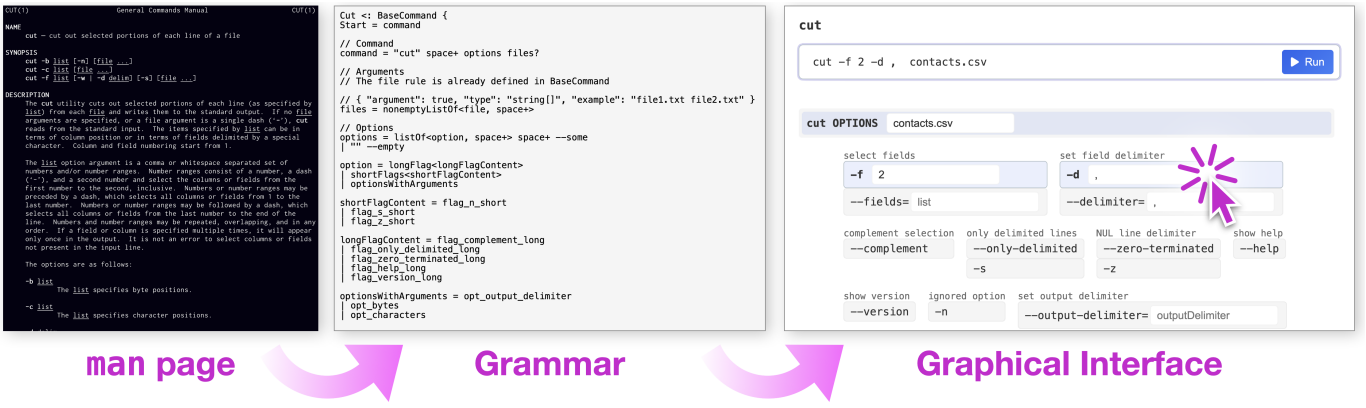

Fig. 1: GUIde automatically provides GUI interfaces for command line tools by translating `man` pages into specifications.

***Abstract*—Although birthed in the era of teletypes, the command line shell survived the graphical interface revolution of the 1980's and lives on in modern desktop operating systems. The command line provides access to powerful functionality not otherwise exposed on the computer, but requires users to recall textual syntax and carefully scour documentation. In contrast, graphical interfaces let users organically discover and invoke possible actions through widgets and menus. To better expose the power of the command line, we demonstrate a mechanism for automatically creating graphical interfaces for command line tools by translating their documentation (in the form of man pages) into interface specifications via AI. Using these specifications, our user-facing system, called GUIde, presents the command options to the user graphically. We evaluate the generated interfaces on a corpus of commands to show to what degree GUIde offers thorough graphical interfaces for users' real-world command line tasks.**

I. Introduction

Since at least the dawn of time-sharing mainframes in the early 1960's [1], the command line interface (CLI) let users run programs on a computer interactively: type out a command on a keyboard (originally an electric typewriter) and, when completed, see its result (typed back on the paper or shown on an electric display). Although an interface conceived for teletype machines in an era of glacial computing speeds, the command line interface endured through the graphical personal computer revolution of the 80's and lives on in modern desktop operating systems—Windows, MacOS, and Linux all include a terminal shell.

*Equal contribution.

However, the command line is no longer considered the normal way to operate a computer, as graphical user interfaces (GUIs) have advantages over CLI apps [2], such as a reduction in cognitive load by shifting from recognition to recall [3]. Nevertheless, programmers still use and write command line tools, in part because CLI applications typically provide more expressive power and because building an entire GUI for *e.g.* a simple script is a non-trivial undertaking.

But even if writing non-GUI tools is easy for the programmer, using them is hard: one must already know the command name, its flags, and its argument structure. Large language models can help somewhat, lettings users write in natural language [4], but AI chat interfaces do not directly help users explore the breadth of all the options of what a command can do. *Discoverability* of options, allowing users to explore, and *interactive tweaking*, allowing users to quickly change properties by clicking, are two key advantages of graphical interfaces not provided by even an AI-augmented command line. Can we "drag Unix into the 80's" [5] by providing graphical interfaces for command line tools? This might provide the best of both worlds: the expressive range of CLI tools along with the usability of graphical widgets.

Related Work

Several systems provide such graphical front-ends. Bespoke [6] automatically generates composable GUIs supporting a subset of a command's options by observing user demonstrations. This interaction model lets users flexibly tweak options, but assumes they have the expertise to supply

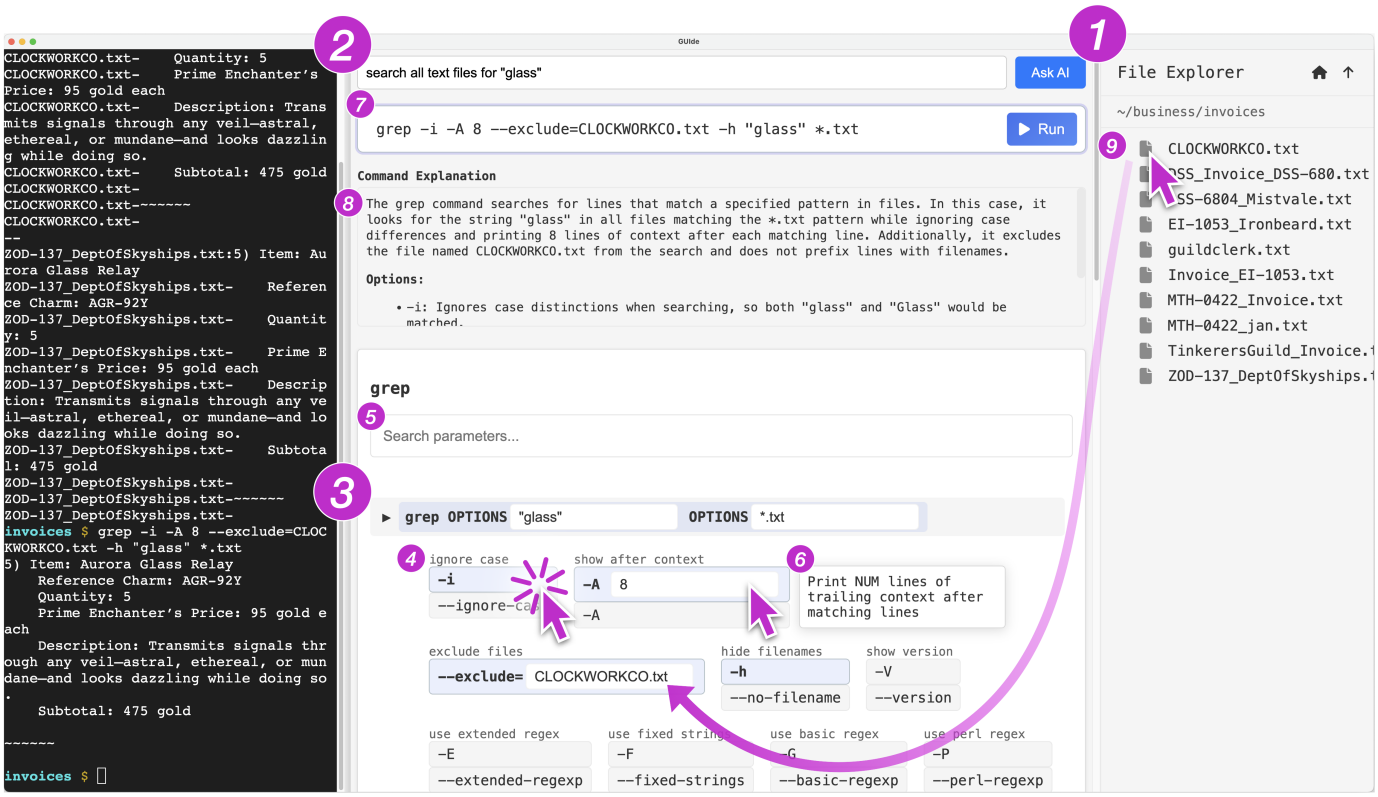

Fig. 2: GUIDE interface, shown for a `grep` command.

a valid command to begin with, limiting its effectiveness for novices. In the early 1990's, Apple's Commando [7] showed graphical dialogs for common Unix commands. These *purpose-built* GUIs streamlined CLI operations, but were not easily adaptable to commands unsupported by the system.

Another approach is to provide more *general-purpose* GUIs. PowerShell's Show-Command is one such example [8]. Given a structured specification, Show-Command generates a GUI that constructs and executes any PowerShell command, including user-defined ones. While Show-Command lowers the barrier of entry for novice users, it relies on a structured, machine-readable specification to generate GUIs. Many traditional Unix tools are, however, documented only through natural language `man` pages, which would need to be manually translated to a more structured form in order to be used with such a system.

While each of these systems does provide a GUI to construct commands, they are *unidirectional*, and only support generating a text command from a GUI. *Bidirectional* interfaces bridge the gap between graphical and textual programming workflows [9]–[11] by letting users modify either the textual or graphical representation, keeping the two in sync [12]–[14].

Large language models (LLMs) also hold potential for automatically generating GUIs, especially in light of recent improvements in LLM agents for software engineering [15]–[19]. Existing approaches to generating GUIs using LLMs, such as in Biscuit [20] and DynaVis [21], create widgets based on user input. Because these systems are reactive, the generated GUIs are small and focused: only options related to the user's interaction are exposed for manipulation. Limiting displayed options can reduce cognitive load, presenting users with a clear subset of options, but does so at the expense of discoverability.

Our Approach: GUIs from Man Pages via AI

Instead, we aim for a *maximal* approach, to show an exhaustive list of a command's parameters. To support both extant and new commands, we aim for *automatic* generation of GUIs from natural language documentation, specifically, from the `man` pages (manual pages) usually provided with a command. We prompt an AI with a `man` page and have it output a grammar describing valid flags and arguments, which is then presented to the user in our graphical application called THE COMMAND LINE GUIDE, hereafter GUIDE. GUIDE offers the user a bidirectional interface for authoring a command: the selected options in the GUI update live as they type a command, and, in the reverse, interaction with the GUI edits the command. This immediate bidirectionally contrasts with *e.g.* PowerShell's Show-Command, which is only a one-shot generator. Specifically, we contribute:

- GUI inference: generating GUI specifications for command line utilities, given only a `man` page.
- GUIDE: a bidirectional GUI-and-text terminal application for authoring and running commands.

Below, we introduce GUIDE through an example, detail the GUI inference process, and then critically evaluate the automatically-generated GUIs for 20 common commands.

II. Command Line GUIde Example

John, a novice command line user, is looking for pricing information about an item for his business, but needs to search multiple files at once.[1][2] He knows he can likely do this with command line tools, and he starts up GUIde. GUIde, shown in Fig. 2, presents a terminal at left, a command editor in the middle, and a file explorer at right.

File explorer. The *file explorer* (1) lets John click to navigate to the directory with all his invoices. The terminal runs the needed `cd` commands automatically.

AI command generation. GUIde provides an AI prompt box (2) for generating and editing commands, similar to recent AI terminal apps [4]. John enters the prompt "`search all text files for "glass"`". The AI produces the command `grep "glass" *.txt`, but when John runs it there are no results!

Flag discoverability and selection. John wonders if `grep` is misconfigured. GUIde creates a graphical interface (3) for editing the command without requiring John to look elsewhere for documentation. The interface offers alternative command forms (here hidden behind a disclosure triangle) as well as a comprehensive list of supported flags for `grep`. This lets John *discover* relevant flags. Scanning through them, he notices the `-i` flag, labeled "ignore case", and wonders if `grep` is case-sensitive by default. John clicks `-i` to *toggle* the `-i` flag (4), adding it to his command. Upon re-running, he now sees the name of the item, "Aurora Glass Relay", in two invoices. But, he cannot see the items' prices. `grep` only shows the lines that match the search string, but the prices are on nearby lines.

Wondering if there is a way to show surrounding lines, John enters "line" in the *parameters search box* (5) and sees an `-A` flag labeled "show after context". When he hovers his mouse over the flag, a tooltip (6) says "Print NUM lines of trailing context after matching lines", which is what he wants. He toggles `-A` on, and fills in its input box with 3.

Bidirectional editing. After running the command, he sees 3 is not enough lines to show the price. He could change the number in the same input box, but GUIde also supports bidirectional editing: the draft command (7) is text-editable. John changes the 3 to an 8 in the full command text, and the GUI below updates to match automatically. Now when John runs the command, the prices he wants are displayed.

Real-time AI Explanation. As John crafts his command, GUIde live updates an AI summary (8) of what the command is expected to do. This helps John build confidence that the command will do what he wants.

File drag-and-drop. John realizes his search includes an old invoice that should be excluded. Instead of manually figuring out the exact path to exclude, John toggles on the `--exclude` flag and simply drags and drops (9) the unwanted file into the flag's text box, which fills in the text box with the file name. The command is updated with the proper exclusion syntax. John re-runs the command and inspects the output for the latest price he paid for Aurora Glass Relays.

Recap. GUIde streamlines the construction of terminal commands with GUI conveniences. A *file explorer* lets users navigate the filesystem and *drag-n-drop files* into command arguments. GUIde lists command flags in a graphical interface, facilitating *discoverability*, and simplifying flag selection by offering *search* and quick *click-to-toggle* to add and remove flags. Editing is *bidirectional* when users want to text-edit the full command rather than use the GUI. And *real-time AI explanations* increase user confidence in their command.

III. Implementation

GUIde's interfaces are generated based on command-specific grammars that describe valid commands. Although developers could write these grammars by hand, we aim to generate GUIs automatically. We prompt an LLM with a command's `man` page to generate the grammars. Surprisingly, although most commands are simple, we found that naïvely prompting LLMs produced unusable grammars with many errors. Below we discuss the more involved prompting and repair process we devised to produce usable grammars, with additional details in a technical supplement [22].

Generating GUIde*-lines from* `man` *pages*

The goal is to produce what we call a "GUIde-line", a context-free grammar (in Ohm [23]) with extra annotations to support GUI rendering. Rules in the grammar can be optionally annotated as representing either a *flag* or an *argument*:

- A *flag* is an optional chunk, *e.g.* `-a` or `--num=10`, that can be toggled on/off in the generated GUI. Whether flags require one, two, or no leading dashes is *not* hard-coded into GUIde, it is based on the grammar structure.
- An *argument* is a chunk for user input, rendered as an input box in the GUI, *e.g.* the three boxes in `cut -d , -f 2 file.csv` are arguments. Note that `-d ,` and `-f 2` are arguments nested *inside* flags.

We use a large language model[3] (LLM) to generate GUIde-lines using the process outlined in Figure 3. First, we prompt an LLM to generate a test suite containing valid invocations of a command based on its `man` page. We then use the test suite and `man` page to generate a draft GUIde-line. Finally, we use LLM agents to correct syntactic errors, lint the GUIde-line, and to fix failing test cases.

Test Suite Generation

To make a test suite, we ask the LLM to generate 10 valid invocations of a command, then to generate a further 10 tests asking it to improve the variety of test cases along several dimensions, including the syntax used to pass arguments, the number of arguments, and use of variables in arguments. Each of the 20 test cases consists of the text

[1] Mythical invoices for the example are AI-generated (OpenAI `gpt-4.1`).

[2] Example video: https://zenodo.org/records/16749005/files/demo.mp4 [22]

[3] `claude-3-7-sonnet-20250219` temperature 1 with thinking tokens

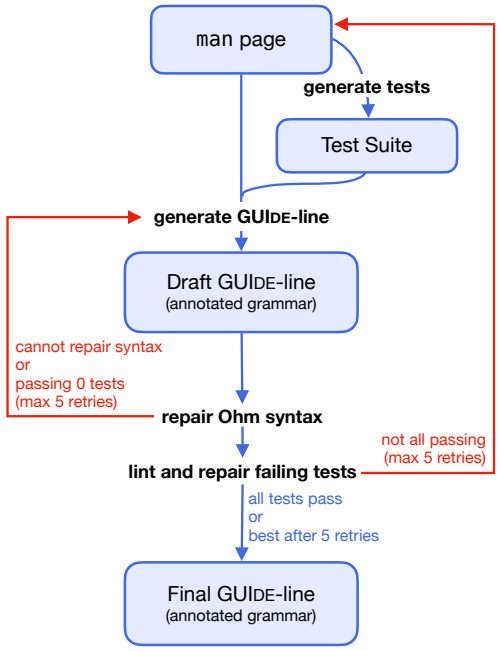

Fig. 3: Automatically creating a GUIDE-line from a `man` page.

to parse (e.g. "`ls -lah`") and flags that are expected (e.g. "`-l`", "`-a`", and "`-h`"). For a test to pass, the grammar must parse the command successfully *and* the parse tree produced must contain nodes for each of the expected flags. The latter condition causes a test to fail if the GUIDE-line contains overly permissive rules that consume more than one flag.

Draft GUIDE*-line Generation*

We prompt the LLM to write an annotated Ohm [23] grammar, providing (1) the `man` page, (2) the generated test suite, (3) three few-shot [24] examples detailing ideal output for the `ln`, `mdfind`, and `nl` commands (showing various top-level forms, flag formats, and formats shared among multiple flags), and (4) several pre-written grammar rules to parse numbers, string literals, embedded commands, flags, etc. This produces a draft GUIDE-line.

Repair with LLM Agents

Draft GUIDE-lines often have errors which make them unusable as-is. We use LLM agents [15]–[19] to repair them. An LLM agent is a prompt that is run in a loop. At each step, the LLM is provided the current GUIDE-line and error messages, and then performs *actions* to debug the issue or edit the GUIDE-line. The loop terminates after a maximum number of iterations, exiting early if the agent achieves its goal. Each agent is allowed to perform a subset of the following actions:

- `replace(diff)` edits the GUIDE-line, applying the diff to the first matching search string, as in other software development agents [15], [19].
- `read()` returns the current GUIDE-line and error message. This helps the agent see the current file state instead of guessing it from the diffs it applied.
- `parse(example, ruleName)` attempts to parse the string `example` under the grammar's `ruleName` in the current GUIDE-line. This lets an agent debug a problem or test if the problem is fixed.
- `finish()` marks the task as complete and exits the loop, allowing the agent to decide when it is done.

We run three agents in series. First, the *syntax repair agent* fixes syntax errors (*i.e.* invalid Ohm grammars) using `read` and `replace`. It is given troubleshooting instructions with suggestions about how to fix common syntax errors. The agent may perform up to 10 actions. If the agent cannot produce a valid grammar, or the grammar it produces passes zero test cases, we regenerate the GUIDE-line (maximum of 5 retries).

Next, the *linter agent* repairs *sequencing errors*. These arise because parsing expression grammars [25], like Ohm, parse alternations (*i.e. or*-clauses) greedily, always taking the first matching rule. Thus, longer rules must precede their prefixes to be matched (e.g. `--print0` must precede `--print`). The linter is instructed to identify and fix sequencing errors, using the `parse` action to test rules and `replace` to fix them. It may perform up to 10 actions or `finish` early.

Finally, the *test case repair agent* fixes failing tests. As with the linter agent, it uses `parse` and `replace` to debug and fix errors (up to 30 actions). It is run once for each failing test case, and is directed to fix the test and instances of the same issue in other parts of the GUIDE-line. After each run, the edited GUIDE-line is tested against the full test suite, replacing the previous version if it has *strictly fewer* failing tests.

If less than 20 of the test cases pass, the *entire* process above starts over with generating new test cases (maximum of 5 retries). Except for unusually complicated commands, no retries are necessary and the above process produces a GUIDE-line that can parse all 20 test cases.

Bidirectional UI from a GUIDE*-line*

When making the GUIDE-line, the LLM annotates which grammar rules represent flags and arguments. Flag annotations include an identifier (to indicate that *e.g.* `-h` and `--help` are the same), a short description (displayed in the main UI), and a longer description (displayed in the tooltip). An argument annotation simply marks a rule as an argument, *i.e.* something to be replaced with a text box. The UI displays the grammar rule name as a placeholder in the text box.

To generate the GUI structure, GUIDE walks the hierarchy of grammar rules, flattening productions into a two level hierarchy: top-level alternatives representing different forms of a command (hidden behind a disclosure triangle in Fig. 2 ❸), and a set of flags for those alternatives, with equivalent flags grouped together (*e.g.* `-V` and `--version`). This conversion from GUIDE-line to GUI is done purely symbolically, without any prompting of an LLM or user intervention.

In the generated GUI, on user interaction the current GUI state is serialized to generate the textual command. To provide bidirectional editing, when the user edits the textual command, GUIDE parses it with the grammar and matches the parse tree nodes with the GUI elements to update the GUI state.

Command	# Recreatable	# Examples	Parse Rate	
sudo	10	176	100.0%	
xargs	9	849	100.0%	
echo	10	344	98.6%	
ssh	10	113	98.2%	
mkdir	10	82	97.6%	
cut	10	189	97.4%	
tr	10	117	97.4%	
ls	10	107	96.3%	
wc	10	27	96.3%	
grep	8	611	95.7%	
dirname	10	64	95.3%	
cat	10	183	95.1%	
tee	10	81	93.8%	
sort	10	188	90.4%	
split	8	78	85.9%	
find	0	5162	81.7%	
rsync	8	125	80.8%	
uniq	10	22	77.4%	
tail	7	65	64.6%	(96.9%)
head	10	70	52.9%	(97.1%)
Mean	*9.0*		*89.8%*	
Total		*8653*		

TABLE I: Evaluation metrics for 20 common commands. Parse rates in (parens) are for manually repaired GUIDE-lines.

IV. Evaluation

To evaluate the automatically-generated GUIDE-lines and their UIs, we tested each GUIDE-line on two metrics—*parseability*, which tests the grammar itself; and *recreatability*, which tests the whether the generated UI is sufficient to invoke a desired command. For our corpus, we use the NL2Bash [26] dataset, which contains bash commands scraped from various online sources. We generated GUIDE-lines for the 20 most commonly occurring commands in NL2Bash listed in Table I (after splitting pipes into separate commands; we also removed I/O redirects and environment variables). We used `man` pages from GNU coreutils for applicable commands, and use `man` pages from Ubuntu 22.04.5 LTS for the remaining commands (`grep`, `rsync`, `ssh`, `sudo`, and `xargs`).

Parseability. A test command is *parseable* if the GUIDE-line parses it successfully. "Parse Rate" in Table I is the percentage of parseable invocations. We deduplicate repeated invocations.

Most GUIDE-lines had a parse rate of over 90%. However, 6 commands fell below this mark. `head` and `tail` used flags not noted in their `man` pages. Both support a numerical flag (e.g. `-8`) to specify the number of lines to show. As this flag is well-known, it merits being manually added to the GUIDE-lines. Doing so brings their parse rates above 90%. `find`'s grammatical structure is highly complex, as it allows for writing nested expressions and boolean logic with its flags. The LLM struggled with `rsync`'s exceptionally long `man` page. `split`'s GUIDE-line had mistakes relating to 2 specific flags, accidentally requiring `--lines` to take a numerical argument when it could also take a variable, and `--filter` mistakenly consumes all flags came after it. `uniq`'s 5 failing examples contained uncommon shorthands and argument-passing formats not described in its `man` page.

Recreatability. We consider an invocation *recreatable* if an equivalent command can be reproduced solely by interacting with the GUI (Fig. 2 ❸), without typing anything in the full command text box (Fig. 2 ❼) other than the command name to pull up the appropriate GUI. For each command, we randomly sampled 10 parsable invocations and clicked flags and filled in arguments to re-create an equivalent invocation. "# Recreatable" in Table I shows the number of successes.

Most simple commands are recreatable. Some commands like `rsync -rvv` are not supported, as the UI only supports using a flag once. The LLM-generated grammar for `tail` conflated flags and positional arguments, which was fine for parsing but confused the GUI generator so the user could not supply *both* flags and a file name argument in the same invocation. The worst GUI was for `find`: its complex query syntax was represented by a grammar which the GUI generator interpreted as over 1000 top-level command forms, resulting in an unusable UI.

V. Discussion

GUIDE works well for ordinary, simple commands. By using arbitrary grammars, we support, *e.g.*, the user typing `ls -lah` and these short flags correctly toggling the `-l`, `-a`, and `-h` flags in the GUI. Nevertheless, GUIDE currently fails for commands that cross some threshold of complexity, such as `find` which embeds its own query language. Further exploration is needed to determine what level of complexity leads to the kinds of breakdowns we observed.

Using arbitrary grammars highlighted to us that the generated GUI is some level of flattening along a continuum. At one extreme, each GUI element would correspond precisely to one grammar rule, supporting the full grammar but requiring the user to navigate a dizzyingly deep nesting of widgets to find the desired options. At the other extreme, the grammar might be fully concretized into all of its (infinitely many!) possible string productions and the user picks from a (infinitely long!) flat list of possible commands. We chose some midpoint on that continuum and found that, while supporting conventional command structures, our midpoint did not scale to *e.g.* `find`.

One strategy for improvement may be to incorporate an additional feedback loop in the generation process: an automatic evaluation of the grammar-generated GUI. Our automatic feedback only looked at test case parse rate, which can work *against* a nice UI: writing a grammar to support odd edge cases and redundant (but allowed) command forms complicates the grammar, adding rules that lead to extraneous GUI elements.

In conclusion, we can generate GUIs from documentation (`man` pages) with LLMs, using annotated context-free grammars as the link between text and graphical interface, suggesting the possibility of generating GUIs for other ad-hoc structured text, such as configuration languages, industry-specific data formats, and other bespoke formats.

—

See the companion supplement [22] for a video demo, source code, and further details on AI grammar generation and GUI generation from a grammar.

Acknowledgments

This work was supported by U.S. National Science Foundation Grants No. 2432644 (*Direct Manipulation for Everyday Programming*) and No. 2107397 (*Human-Centric Program Synthesis*).

References

[1] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley, "An Experimental Time-sharing System," in *Proceedings of the 1962 spring joint computer conference, AFIPS 1962 (Spring), San Francisco, California, USA, May 1-3, 1962*, 1962, https://doi.org/10.1145/1460833.1460871.

[2] E. L. Hutchins, J. D. Hollan, and D. A. Norman, "Direct Manipulation Interfaces," *Hum. Comput. Interact.*, vol. 1, no. 4, pp. 311–338, 1985, https://doi.org/10.1207/s15327051hci0104_2.

[3] J. Neilsen, *Heuristic Evaluation*. USA: John Wiley & Sons, Inc., 1994.

[4] "The intelligent terminal," https://www.warp.dev/.

[5] S. Kell, "Dragging unix into the 1980s (and beyond?) - liveness and source-level reflection," Curry On!, 2019. [Online]. Available: https://www.youtube.com/live/nwrCestQTaw

[6] P. Vaithilingam and P. J. Guo, "Bespoke: Interactively Synthesizing Custom GUIs From Command-Line Applications By Demonstration," in *Symposium on User Interface Software and Technology (UIST)*, 2019, https://doi.org/10.1145/3332165.3347944.

[7] A. Rosen, J. Pittelkau, and The MacUser Labs Staff, "The Best of UNIX and the Mac: A/UX 2.0," *MacUser*, January 1991, https://archive.org/details/MacUser9101January1991/page/n119/mode/2up.

[8] Microsoft, "Show-command," 2024. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/show-command

[9] B. Shneiderman, "Direct manipulation: A step beyond programming languages," *Computer*, vol. 16, no. 08, pp. 57–69, 1983.

[10] M. Read and C. Marlin, "Generating direct manipulation program editors within the multiview programming environment," in *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops*, 1996, pp. 232–236.

[11] S. P. Reiss, "Graphical program development with pecan program development systems," *ACM SIGSOFT Software Engineering Notes*, vol. 9, no. 3, pp. 30–41, 1984.

[12] B. Hempel, J. Lubin, and R. Chugh, "Sketch-n-Sketch: Output-Directed Programming for SVG," in *Symposium on User Interface Software and Technology (UIST)*, 2019.

[13] R. Schreiber, R. Krahn, D. H. Ingalls, and R. Hirschfeld, *Transmorphic: Mapping direct manipulation to source code transformations*. Universitätsverlag Potsdam, 2017, vol. 100.

[14] C. Omar, D. Moon, A. Blinn, I. Voysey, N. Collins, and R. Chugh, "Filling typed holes with live guis," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 511–525.

[15] J. Yang, C. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," *Advances in Neural Information Processing Systems*, vol. 37, pp. 50 528–50 652, 2024.

[16] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 819–831.

[17] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying LLM-based Software Engineering Agents," *CoRR*, vol. abs/2407.01489, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2407.01489

[18] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou, "Large language model-based agents for software engineering: A survey," 2024. [Online]. Available: https://arxiv.org/abs/2409.02977

[19] "Cline - AI autonomous coding agent for VS code," https://cline.bot/.

[20] R. Cheng, T. Barik, A. Leung, F. Hohman, and J. Nichols, "Biscuit: Scaffolding llm-generated code with ephemeral uis in computational notebooks," in *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2024, pp. 13–23.

[21] P. Vaithilingam, E. L. Glassman, J. P. Inala, and C. Wang, "DynaVis: Dynamically Synthesized UI Widgets for Visualization Editing," in *Conference on Human Factors in Computing Systems (CHI)*, 2024, https://doi.org/10.1145/3613904.3642639.

[22] S. R. Kasibatla, K. Medleri Hiremath, R. Rothkopf, S. Lerner, H. Xia, and B. Hempel, "The Command Line GUIde: Graphical Interfaces from Man Pages via AI Supplementary Materials," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.16749004

[23] A. Warth, P. Dubroy, and T. Garnock-Jones, "Modular semantic actions," *ACM SIGPLAN Notices*, vol. 52, no. 2, pp. 108–119, 2016.

[24] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[25] B. Ford, "Parsing expression grammars: a recognition-based syntactic foundation," in *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2004.

[26] X. V. Lin, C. Wang, L. Zettlemoyer, and M. D. Ernst, "Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system," *arXiv preprint arXiv:1802.08979*, 2018.