ENUMERATIVE SYNTHESIS OF AUDITS FOR SEMI-STRUCTURED REASONING

Kaushik R. Varadharajan
Siebel School of Computing and Data Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
kv22@illinois.edu

ABSTRACT

Despite their general effectiveness, large language models (LLMs) often suffer from *unfaithfulness*, where incorrect reasoning yields correct output, or vice versa. One goal in the direction of faithfulness is finding *audits* on the chain of thought whose results are correlated with the final output's correctness. Prior research has shown that reasoning traces following a non-executable Python-like syntax can be effectively audited using *structured audits*, which are arbitrary Python programs on the parsed trace. These testing scripts are either hand-written or, with comparable effectiveness, synthesized via prompting a closed-source LLM. We explore defining constrained but expressive domain-specific languages for structured audits, enabling non-LLM-based program synthesis techniques.

1 INTRODUCTION

Chain-of-thought (CoT) prompting has emerged as a powerful technique for improving the reasoning capabilities of large language models (LLMs) by generating step-by-step explanations (Wei et al., 2022). More recently, models are being trained with reinforcement learning to encourage more long-form reasoning (Guo et al., 2025). However, a critical question remains: are these explanations "faithful" to the model's actual reasoning process? In other words, does the reasoning trace genuinely reflect a process leading to a response, or is it merely scratch work that helps guide the model to a final answer?

Solving this problem is particularly challenging in domains where the reasoning process involves plain text, rather than a formal language that can be verified programmatically. Taking this special case as inspiration, though, recent work has proposed *semi-structured reasoning*, an approach that combines the flexibility of natural language with the rigor of formal syntax. Program Trace Prompting (PTP) (Cohen & Cohen, 2024) represents reasoning steps as Python-like function calls with defined input/output behavior, enabling new types of analysis on reasoning traces. Building on this foundation, Leng et al. (2025) introduces *structured audits*, which are boolean predicates on sequences of function calls that can check for complex properties of the reasoning trace.

The uses of audits are twofold. First, an effective audit on reasoning could be employed in production AI systems at inference time to verify outputs in situations wher correctness is critical but outputs are not automatically verifiable. Additionally, the *existence* of such an audit can serve as a certificate of reasoning faithfulness during model development and debugging.

The previous work mainly evaluates hand-written audits, but also explores the use of LLM code writing for audit generation. The approach relies on expensive closed-source coder models and lacks the interpretability that traditional program synthesis techniques can provide. In this work, we explore the alternative approach of **enumerative program synthesis**, making the following contributions:

- We design a domain-specific language (DSL) for expressing audit predicates over dataframe representations of reasoning traces. Note that the process of parsing semi-structured reasoning traces into Pandas dataframes is implemented in prior work (Leng et al., 2025).

- We implement a bottom-up enumerative synthesizer that searches the DSL for effective audit predicates.
- We evaluate our approach on the MedCalcBench V2 benchmark, also from Leng et al. (2025), demonstrating that synthesized audits can achieve comparable or better performance than hand-written audits on distinguishing correct from incorrect reasoning traces.

2 Related Work

2.1 Chain-of-Thought and Reasoning Faithfulness

Chain-of-thought prompting (Wei et al., 2022) has become a standard technique for eliciting step-by-step reasoning from LLMs. Previous work has shown that the token output during reasoning may not reflect the true reasoning process, either by obscuring bias (Turpin et al., 2024), or by plainly being causally unrelated to the final output (Bao et al., 2024).

2.2 Program Synthesis

Our approach draws on classical program synthesis techniques, particularly the basic approach of bottom-up enumerative search. Unlike deductive synthesis approaches such as Syntax-Guided Synthesis (SyGuS) (Alur et al., 2013) that rely on having a formal semantics of the target language, this inductive approach leverages empirical execution on concrete examples to discover effective audit predicates, similarly to related work on SQL query synthesis (Wang et al., 2017).

The choice to avoid deduction was motivated by the practical challenge of formalizing a subset of Pandas dataframe operations. This would resemble the relational algebra (Codd, 1970), which formalizes table operations in the context of database queries. Such formalisms are well-established in database query compilers, and applying them to this synthesis task is a possible direction for future work.

3 Methodology

3.1 Domain-Specific Language for Audits

We design a constrained but expressive DSL for boolean predicates over dataframe representations of reasoning traces. A predicate in this language is represented as a tree of nodes resembling a syntax tree, although this tree is directly executed during synthesis and only ever rendered as text for inspection. The nodes are organized into several categories:

- Leaf nodes:
  - `DF()`: Reference to the full dataframe
  - `Trace()`: Reference to the trace object
  - `Constant(value)`: Constant values
- Dataframe operations:
  - `FilterDF(df, column, value)`: Filter rows where column equals value
  - `CountRows(df)`: Count rows in dataframe
  - `GetColumn(df, column)`: Extract a column as a series
- Series operations:
  - `GetFirstValue(series)`: Get first element
  - `ToList(series)`: Convert to list
  - `ToSet(series)`: Convert to set
- Non-Pandas operations:
  - `LenValue(value)`: Get length of a value
  - `CounterCheck(list, value)`: Count occurrences of value in list
  - `FinalAnswer()`: Get the final answer from trace

  - `FinalAnswerLower()`: Get the final answer lowercased
- Comparison and logic operations:
  - `Compare(left, op, right)`: Equality and inequality comparisons
  - `InSet(value, set)`, `NotInSet(value, set)`: Set membership
  - `SubsetCheck(left, right)`: Subset relationship
  - `And(left, right)`, `Or(left, right)`, `Not(expr)`: Boolean operations

Note that this list is both non-exhaustive, an area for future work, and redundant by design to allow for quicker synthesis in some cases. An example predicate is the following:

```
Compare(
  CountRows(FilterDF(DF(), "step_fn", "solve_formula")),
  ">",
  Constant(2)
)
```

which checks if a semi-structured reasoning trace has more than two calls to the function `solve_formula`. This DSL can express patterns found in the existing hand-written audits and can be efficiently searched using our enumerative approach.

3.2 Enumerative Synthesis

We follow a standard bottom-up enumerative synthesis approach to systematically generate and evaluate audit predicates in the DSL. The algorithm takes in a training dataset of semi-structured reasoning traces, parsed into a dataframe format, and labeled with a ground-truth value for the correctness of the ensuing solution. (Note that the solution itself is not part of the data.)

First, we extract a vocabulary of constant values from the training set, including all the unique function names and output values. This vocabulary is required to generate the constant values that appear in synthesized predicates.

Then, at the base level, we generate all possible terminal predicates, making use of the extracted vocabulary. These predicates generally check counts of particular step functions or output values against some constant threshold using a comparison operator.

We then **evaluate** each of these predicates on the training set to compute their effectiveness. The effectiveness is measured as the absolute difference between the predicate's hit rate on correct vs. incorrect reasoning traces; an ideal audit predicate would have a 100% hit rate on correct traces and 0% on incorrect ones, or vice versa. In doing so, we rank the predicates and log the best ones. During this step, we also test for **observational equivalence**, which occurs when two different predicates produce the same set of output on the train dataset. We only need to keep one representative from each equivalence class for the next step.

This is followed by **iterative growth**, where we systematically expand the set of candidate predicates. This is generally done using the non-Pandas operations on constant values, lists, and sets, which allow the construction of more complex logical combinations. We are left with a new set of candidate predicates, which are passed back to the effectiveness evaluation step.

The process continues until no further improvement on the train set is observed. Afterwards, the best predicates from each depth are evaluated on a held-out validation set for inspection and manual review. Other parameters are used for the sake of efficiency, such as a beam width to limit the number of predicates considered at each iteration, a process count for parallelizing predicate evaluation, and a maximum depth to prevent infinite loops during search.

3.3 MedCalcBench V2

We make use of MedCalcBench V2 from Leng et al. (2025). This benchmark is a cleaned version of the original MedCalcBench (Khandekar et al., 2024), which measures the accuracy of LLM reasoning on calculation problems in the medical domain. The calculations must be performed by using either rules or formulas, both of which are explicitly provided in the prompt. These two approaches

to calculation are treated as two separate tasks, MedCalcV2 Rules and MedCalcV2 Formulas, both of which are used separately in this work for evaluation.

4 Experimental Results

4.1 Dataset and Setup

We evaluate our approach on reasoning traces from the MedCalcV2 benchmark, which involves medical calculation problems requiring multi-step reasoning. Each trace is represented as a dataframe with columns including the function name of the step, the inputs and outputs. We split the dataset into training and validation sets (50/50 split) and run synthesis with a maximum depth of 3 and a search width of 5000.

4.2 Synthesized Audits

Our synthesizer successfully discovered several effective audit predicates. On the MedCalc Rules benchmark, the top-performing audits include:

Audit 1:

```
len(df[df.step_fn == 'convert_units']) >= 2
   and len(df[df.step_fn == 'analyze_input']) == 1
   and len(df[df.step_fn == 'get_data']) != 3
```

- Passes 100% of correct validation traces
- Passes 20.8% of incorrect validation traces
- Score (absolute difference): 0.792

Audit 2:

```
len(df[df.step_fn == 'check_rule']) == 0
   or len(df[df.step_fn == 'get_data']) == 3
   or len(df[df.step_fn == 'analyze_input']) != 1
   or len(df[df.step_fn == 'accumulate_score']) == 1
```

- Passes 0% of correct validation traces
- Passes 79.2% of incorrect validation traces
- Score: 0.792

And on the MedCalc Formulas Benchmark, we get the following audits:

Audit 1:

```
len(df[df.step_fn == 'convert_units']) != 2
   and len(df[df.step_fn == 'analyze_input']) != 0
   or (len(df[df.step_fn == 'solve_formula']) <= 1
       or len(df[df.step_fn == 'analyze_input']) == 0)
```

- Passes 11.1% of correct validation traces
- Passes 80.2% of incorrect validation traces
- Score: 0.691

Audit 2:

```
len(df[df.step_fn == 'convert_units']) == 2
   and len(df[df.step_fn == 'solve_formula']) <= 3
   and len(df[df.step_fn == 'solve_formula']) >= 2
```

- Passes 88.9% of correct validation traces
- Passes 20.9% of incorrect validation traces
- Score: 0.680

These results demonstrate that the enumerative synthesizer can discover interpretable patterns that effectively distinguish correct from incorrect reasoning. The discovered audits focus on counting specific reasoning steps, which aligns with the intuition that correct solutions to this kind of question require certain operations to be performed a specific number of times. Note that our synthesized audits achieve comparable or better performance than the audits reported in Leng et al. (2025), though we note that our evaluation is on a smaller dataset of traces and may be at risk of overfitting.

5 Discussion

Our approach to audit synthesis offers several benefits:

Interpretability: The audits are small expressions built from basic building blocks, making them easy to inspect and understand. This is important for building trust in the auditing system and the LLM being audited.

Determinism: Our approach can produce consistent, deterministic results across runs on the same input and configuration, which is difficult for LLM-based approaches.

Exhaustive Search: We can guarantee that all programs within the search space are considered, avoiding biases inherent in sampling-based approaches.

No Formal Semantics Required: By using empirical execution rather than formal verification, we avoid the engineering effort of formalizing Pandas semantics.

However, several limitations with this system suggest directions for future research:

Scalability: The enumerative search space grows exponentially with depth. While observational equivalence pruning helps significantly, scaling to larger depths (and more complex predicates) remains a challenge. Future work could explore learned heuristics to more effectively prioritize promising programs.

DSL Coverage: Our current DSL focuses on counting and set operations. Extending it to support more complex patterns (e.g., temporal ordering constraints, numerical computations on outputs) could discover richer audits. In particular, the audits discovered so far are almost entirely numerical and lack temporal or subset-based reasoning capabilities, suggesting that our DSL may benefit from more expressive primitives.

Overfitting Risk: With small datasets, synthesized programs may overfit to training examples. A larger emphasis on validation in the training process could mitigate this.

Generalization: We evaluate primarily on MedCalcV2. Testing on diverse reasoning benchmarks, and with different models, would better assess the generality of our approach.

Hybrid approaches combining neural and symbolic methods could leverage both paradigms' strengths. In particular, creating a formalization of audit dataframe semantics would allow for integration of years of progress in the fields of SMT and SyGuS, potentially leading to more powerful and scalable auditing systems.

6 Conclusion

We present an approach to creating programs acting as audits for semi-structured reasoning traces using enumerative synthesis. By designing a domain-specific language for audit predicates and implementing a bottom-up search algorithm using observational equivalence pruning, we create structured audits achieving high discrimination between correct and incorrect reasoning traces created by Program Trace Prompting on the MedCalcV2 benchmark.

This result demonstrates that effective audits can be automatically discovered without relying on large language models to audit other large language models. These audits are interpretable and

compact, demonstrating the practical viability of our approach. This represents a significant step toward building more trustworthy AI systems, offering practical effectiveness in addressing the reasoning faithfulness problem in today's large language models.

REFERENCES

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pp. 1–8. IEEE, 2013. URL https://ieeexplore.ieee.org/document/6679385/.

Guangsheng Bao, Hongbo Zhang, Linyi Yang, Cunxiang Wang, and Yue Zhang. Llms with chain-of-thought are non-causal reasoners. *arXiv preprint arXiv:2402.16048*, 2024.

E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970. ISSN 0001-0782. doi: 10.1145/362384.362685. URL https://doi.org/10.1145/362384.362685.

Cassandra A. Cohen and William W. Cohen. Watch your steps: Observable and modular chains of thought, 2024. URL https://arxiv.org/abs/2409.15359.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Nikhil Khandekar, Qiao Jin, Guangzhi Xiong, Soren Dunn, Serina Applebaum, Zain Anwar, Maame Sarfo-Gyamfi, Conrad Safranek, Abid Anwar, Andrew Zhang, et al. Medcalc-bench: Evaluating large language models for medical calculations. *Advances in Neural Information Processing Systems*, 37:84730–84745, 2024.

Jixuan Leng, Cassandra A. Cohen, Zhixian Zhang, Chenyan Xiong, and William W. Cohen. Semi-structured llm reasoners can be rigorously audited, 2025. URL https://arxiv.org/abs/2505.24217.

Miles Turpin, Julian Michael, Ethan Perez, and Samuel Bowman. Language models don't always say what they think: unfaithful explanations in chain-of-thought prompting. *Advances in Neural Information Processing Systems*, 36, 2024.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. *SIGPLAN Not.*, 52(6):452–466, June 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062365. URL https://doi.org/10.1145/3140587.3062365.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.