EXPLORING COMPLEXITY ESTIMATION WITH SYMBOLIC EXECUTION AND LARGE LANGUAGE MODELS

This work explores the symbiosis of symbolic analysis and large language models (LLMs). In particular, we focus on estimating worst-case complexities by expanding the reach of symbolic analysis using LLMs. We attempt to build a closer integration of Symbolic PathFinder (SPF) and models such as ChatGPT. Preliminary results indicate that LLMs can help SPF to amplify its search for inputs which could trigger denial-of-service attacks. The results and insights gained in this work will help researchers and software practitioners to design and develop secure software systems in the future. All datasets and implemented tools will be made open-source following open-source principles.

AUTHORS

Adrians Skapars (University of Manchester)
- adrians.skapars@postgrad.manchester.ac.uk

Youcheng Sun (University of Manchester)
Yannic Noller (SUTD)
Corina Pasareanu (Nasa Ames/CMU)

RELATED WORK

[1] K. Luckow, R. Kersten and C. Păsăreanu, "*Symbolic Complexity Analysis Using Context-Preserving Histories*," 2017 IEEE ICST, pp. 58-68

[2] TDi Wang and Jan Hoffmann, "*Type-guided worst-case input generation*". 2019 Proc. ACM Program. Lang. 3, POPL, Article 13, 30 pages.

RESEARCH QUESTIONS

- Exhaustive search for worst-case constraints is only tractable for smaller inputs. Can LLMs predict what the constraints will be for large inputs?
- If so, can we externalise their internal decision procedure for making these predictions?
- Can LLMs refine their predictions based on some reward mechanism? Can this be automated?
- Can decision procedures be mapped to mathematical generalisations, defining the set of constraints for any input size?
- How best to evaluate solutions to this problem?

①
```
public class SimpleUnique {

    public static void algo(char[] chars) {
        boolean fail = false;
        // Something is done with input variables
        for (int i = 0; i < N-1; i++) { ⋯
        }
        // Which determines whether expensive operation occurs
        if (!fail) { ⋯
        }
    }
```

②
```
CONSTRAINT N/A
in0 != in1
in1 != in2 && in0 != in2 && in0 != in1
in2 != in3 && in1 != in3 && in1 != in2 && in0 != in3 && in0 != in2 && in0 != in1
in3 != in4 && in2 != in4 && in2 != in3 && in1 != in4 && in1 != in3 && in1 != in2 &&
in4 != in5 && in3 != in5 && in3 != in4 && in2 != in5 && in2 != in4 && in2 != in3 &&
in5 != in6 && in4 != in6 && in4 != in5 && in3 != in6 && in3 != in5 && in3 != in4 &&
in6 != in7 && in5 != in7 && in5 != in6 && in4 != in7 && in4 != in6 && in4 != in5 &&
in7 != in8 && in6 != in8 && in6 != in7 && in5 != in8 && in5 != in7 && in5 != in6 &&
in8 != in9 && in7 != in9 && in7 != in8 && in6 != in9 && in6 != in8 && in6 != in7 &&
```

③
```
def generate_constraints(N):
    constraints = []
    for i in range(N):
        for j in range(i+1, N):
            constraints.append(f'in{i} != in{j}')
    return constraints
```

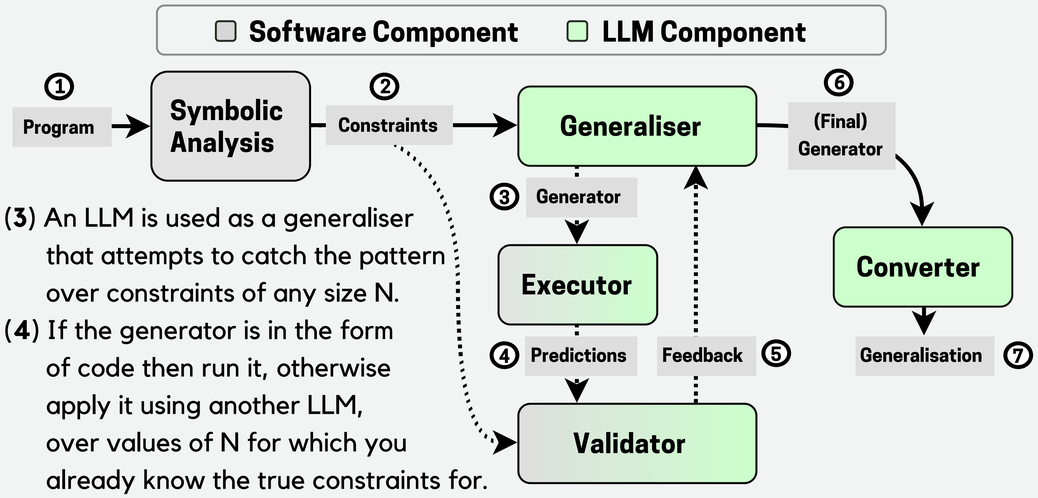
④
```
N=3
in0 != in1, in0 != in2, in1 != in2
N=4
in0 != in1, in0 != in2, in0 != in3, in1 != in2, in1 != in3,
!= in3
N=5
in0 != in1, in0 != in2, in0 != in3, in0 != in4, in1 != in2,
!= in3, in1 != in4, in2 != in3, in2 != in4, in3 != in4
N=6
in0 != in1, in0 != in2, in0 != in3, in0 != in4, in0 != in5,
!= in2, in1 != in3, in1 != in4, in1 != in5, in2 != in3, in2
in4, in2 != in5, in3 != in4, in3 != in5, in4 != in5
```

⑦
```
The python code generates a set of inequalities that stipulate all "in" variables from 0 to "N-1" are pairwise different. The mathematical definition would be:

S = {in[i] ≠ in[j] | 0 ≤ i < j < N}
for all i, j ∈ Z+
```

PROPOSED FRAMEWORK

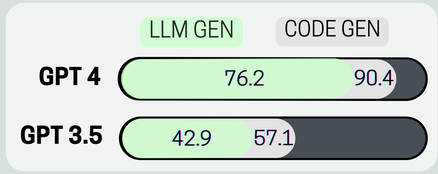
(1) A Java program input to a symbolic analysis tool such as SPF-WCA [1].
(2) Symbolic analysis gets the constraints for symbolic inputs which cause the worst-case execution time of the program, for several smaller input sizes N.



(3) An LLM is used as a generaliser that attempts to catch the pattern over constraints of any size N.
(4) If the generator is in the form of code then run it, otherwise apply it using another LLM, over values of N for which you already know the true constraints for.
(5) If the predictions do not match, give this feedback to the generaliser and repeat.
(6) If the generator can recreate all known and hidden constraints, deem it correct.
(7) Use yet another LLM to convert the generator into a mathematical definition, of a set of constraints parametrized by an arbitrary N, for formal analysis.

METHOD AND PRELIMINARY FINDINGS

- A small dataset of 21 Java programs, with varying complexities in their worst-case conditions, were assembled/ created for evaluation. Generalisers were allowed a maximum of 10 rounds of feedback before being marked unsuccessful.
- The larger GPT 4 performs much better than 3.5, being more receptive of feedback, more consistent in response formatting and offering more creative solutions. This may indicate that the real-world application of this framework only became viable recently.
- Generator evaluation has a surprising amount of influence on performance, expressing gens. as executable Python code being faster to evaluate and leading to higher success rates, possibly due to GPTs extended training on code.
- LLMs can accurately predict the constraints of program inputs over which it could not generalise, indicating the possibility for further improvements.



Generalisation success rate % over dataset, under various framework configurations

FUTURE WORK

- Besides for manual inspection, we currently have limited guarantees that the generalisations output hold for all input sizes, N.
- We plan to use theorem provers to show that our outputs are indeed valid. Alternatively, we may provide extensive statistical verification.
- Future work can benchmark against our results, but we are also interested in comparing against existing methods for finding worst-case inputs.
- Our approach is limited by an LLMs context-window size, which could be partially remedied by normalising constraints to shorter formats.
- We aim to minimise the number of LLM API calls.