Neuro-Symbolic Specification Generation for C programs

George Granberry, Wolfgang Ahrendt, Moa Johansson
Chalmers University of Technology
{georgegr,ahrendt,jomoa}@chalmers.se

Writing formal specifications as e.g. pre- and postconditions should ideally be done before the programmer starts their work. This is however rarely the case, as it can be a difficult and laborious task. The field of formal methods has developed a plethora of symbolic tools for formal analysis of programs. These tools, which employ a range of techniques from both static and dynamic analyses, are traditionally focused on identifying specific subsets of property types. Beyond that, they are not very flexible. Large language models (LLMs), on the other hand, have complementary features. Provided they are adequately trained and prompted, they are less restricted and can potentially generate a variety of program code, specifications etc. However, a challenge is how to prompt them to generate *good quality* annotations that actually express interesting non-trivial properties.

In this research, we introduce a prototype neuro-symbolic specification generation system called ACSLyst, which combines symbolic analyses and LLMs to generate ACSL annotations for C programs. We examine how integrating outputs from symbolic analysis tools from the Frama-C ecosystem into prompts to GPT-4 influences the quality and characteristics of annotations generated. Specifically, we compare the annotations generated when the GPT-4 prompts are augmented with test-cases produced by the PathCrawler tool, and reports generated by the Eva value analysis tool.

When prompting GPT-4 with just the plain program code, it produce a large number of annotations, although many of them are trivial, and does not add much to our understanding of the program. We observed that when including results from symbolic analyses by PathCrawler and Eva into the prompts, the resulting specifications were smaller, and more focused on properties aligned with the symbolic tools. E.g. when given Eva reports, which include runtime alarms, the generated specifications focused much more on mitigated runtime errors. And when PathCrawler test-cases were included, the annotations were skewed towards an abstract view of the program rather than focusing on implementation details.

In summary, we believe this is an interesting field for further exploration of specification generation, leveraging the strengths of both traditional symbolic tools and LLMs. With our eventual goal being to craft a copilot that more effectively incorporates intent and precision across implementation, specifications, and tests, we plan to expand upon this foundational study by developing systems capable of dynamically selecting symbolic tools to integrate with LLMs.