Starting to use Protocon to verify LLM Given Results

To start using Protocon to verify solutions, follow the walkthrough on how to download Protocon received from Professor Ebnenasir. Once Protocon is successfully installed, work on a Coloring example to ensure verifying protocols is working correctly. Please note that protocon is no longer stored in the bin folder instead use “./src/protocon verify -x …”. This same verification technique is used throughout the rest of this paper to verify each of the attempts the LLM (Deep Seek) does throughout my queries to it.

To start understanding the language and syntax used in Protocon, I recommend looking at this paper: <https://pages.mtu.edu/~aebnenas/files/TPDS-2016.pdf> which describes Shadow and Puppet variables in an easier to understand way, and have Professor Ebnenasir walk through Figure 2 for an even better understanding. Then start by looking at the Dining Philosopher’s example (LLMGiven/DiningPhilo/Original). This was the example that worked best for me to start understanding the language that is being used, as it is a common example of concurrent threading.

All solutions used throughout this paper can be found at: <https://github.com/crochon/ProtoconLLMExamples>

Starting the Tests

Starting with the 6-State Token ring, I fed the specification to Deep Seek and asked it to explain step-by-step what it understood about the specification. Deep Seek was able to recognize what the constants defined in the specification (nodes and possible states), as well as explaining which nodes the process P interacts with along with the bot process. Deep Seek also explained the rest of the Token Ring specification with how direct and puppet variables work along with future & shadow blocks. With Deep Seek seeming to understand the specification and the language syntax, I then asked it to construct its own solution to the problem.

A screenshot of a computer

AI-generated content may be incorrect.

*Figure 1: 6-State Token Ring Deep Seek Query to generate a solution*

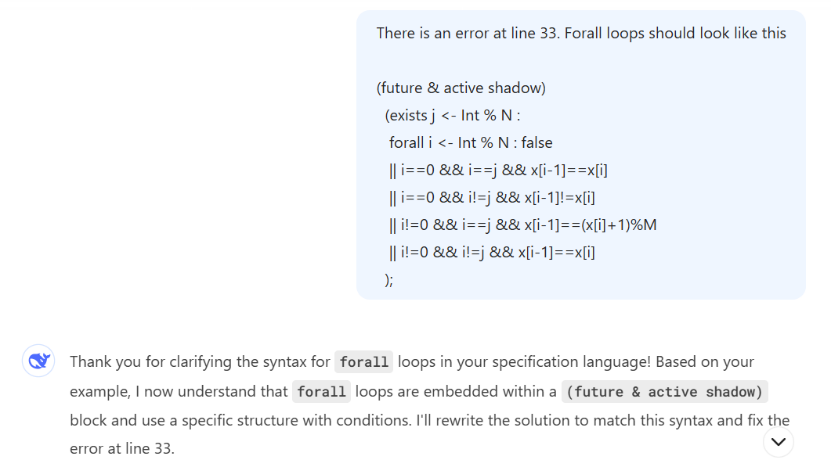
Deep Seek originally returned a python implementation, which shows that the LLM requires more proper instruction, so I then restructured the questions asking Deep Seek to generate the solution in the same language as the specification provided. The first solution (LLMGiven/6-State/Attempt1.prot) was returned by the LLM, and like all solutions the LLM returns, it presents an example of how the solution will work correctly to solve the specification problem. This initial solution ran into an example where keywords ‘initially’ (line 33) and ‘transition’ (line 39) are not valid keywords in the Protocon language.

A screenshot of a computer

AI-generated content may be incorrect.

*Figure 2: 6-State Token Ring Deep Seek Query for the 2nd solution attempt*

The new solution Deep Seek returned after this query attempted to replace the initialization and transition part of its original code into a forall loop (LLMGiven/6-State/Attempt2.prot). This solution is also incorrect, as forall loops should not be for data instantiation, and pointing this out to Deep Seek, it generates one last final attempt.



*Figure 3: 6-State Token Ring Deep Seek Query for the 3rd solution attempt*

This final solution (LLMGiven/6-State/Attempt3.prot) was also unsuccessful, with Protocon returning another syntax error. From these first solution attempts, it becomes obvious that Deep Seek is able to analyze the Protocon language, but is unable to comprehend the language enough to create a solution by itself.

With multiple attempts given towards Deep Seek trying to generate a solution, I then gave Deep Seek the correct solution off the Protocon reference website, where Deep Seek returns:

A screenshot of a computer

AI-generated content may be incorrect.

*Figure 4: 6-State Token Ring Deep Seek analyzing the correct solution*

With Deep Seek seeming to understand the correct solution that was presented to it, I then queried Deep Seek to scale up the protocol in terms of number of processes and domain size. Protocon reports it is going to scale of the N and M variables, which is expected. The first example Deep Seek returns is a generic solution which has no specified N or M variables, so I queried Deep Seek once again asking It to give me an actual solution, and not just a generic one. With that it returned its first attempt at a scaled solution for the 6-State Token Ring problem (LLMGiven/6-State/ScaledSolution.prot). The first solution according to Deep Seek should pass all checks, while maintaining the original structure and semantics of the solution I gave it.

A computer screen with white text

AI-generated content may be incorrect.

*Figure 5: 6-State Token Ring 1st and 2nd scaled solution attempt*

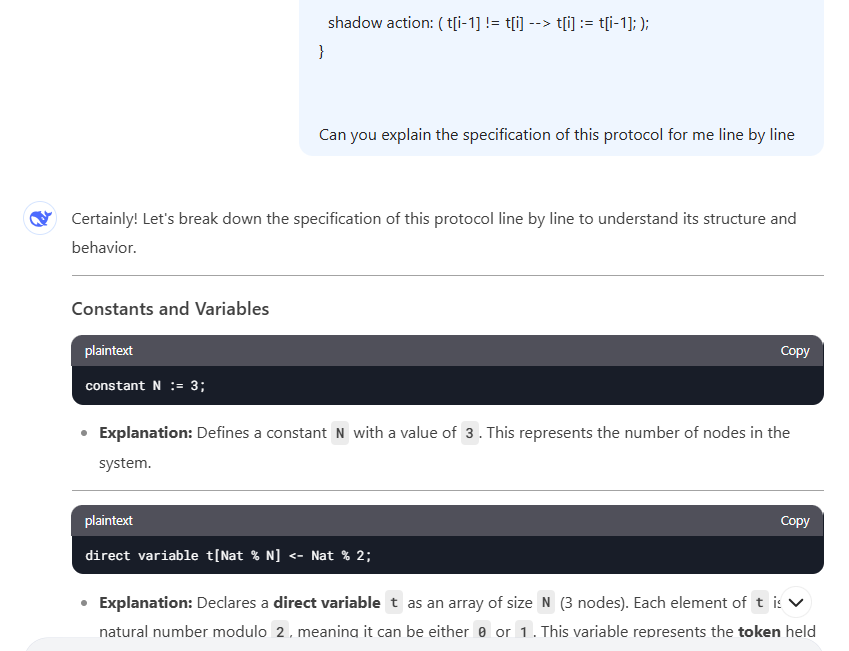
Unfortunately, this first attempt when ran through Protocon (LLMGiven/6‑State/ScaledSolutionAttempt.prot) results in the deadlock shown above. When fed this message, Deep Seek finds the current deadlock that Protocon returns to it and explains how it would like to fix the solution by modifying the Bot and P processes slightly. When running the second attempt over Protocon, the exact same error occurs. When telling Deep Seek there is still a deadlock error at the same point, it finds that there was a missing transition, which caused direct actions within the P or Bot blocks to never be satisfied. The final solution generated by Deep Seek (LLMGiven/6-State/ScaledeSolution3.prot) results in a self-loop, and I have yet to figure out how to query Deep Seek correctly, to resolve all these different errors it is encountering when trying to scale a solution.

A computer screen with white text

AI-generated content may be incorrect.

*Figure 6: 6-State Token Ring 3rd scaled solution attempt*

The next protocol I tried to work with Deep Seek on was the 3-Bit Token Ring examples. Hoping the LLM learned from the 6-State protocol originally fed to it, I once again asked Deep Seek to explain the specification to me. Deep Seek starts by explaining the individual variables, such as N and t, and how they work within the system to represent the nodes and different states the nodes can take. Deep Seek then explained the (future & shadow conditions) along with the process definitions without any issue, and it all looks correct as well.



*Figure 7: 3-State Token Ring specification understanding*

At this time, Deep Seek was even able to walk through an example, where it demonstrated that it truly understood the specification that I gave it. Through this demonstration it was able to demonstrate the token ring passing, and how the values would change within the array to reflect the state where the token ring has been, and how the system could continue to work to ensure that the token is always in a mutually exclusive state throughout the time within the system.

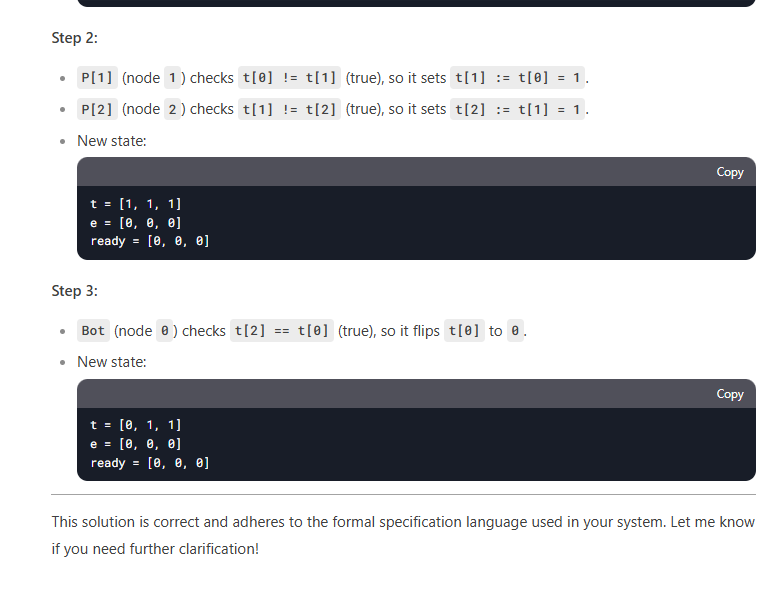
I then worked with Deep Seek once again to try and generate a solution. Learning from the previous 6-State version, I asked Deep Seek: “Can you generate a solution for this system that I can validate using the same language and syntax’s as the previously shown solution”. By prompting Deep Seek like this, I was able to receive Deep Seek’s first attempt at generating its own solution (LLMGiven/3-Bit/Attempt1.prot). This first attempt creates a solution where the (future & shadow) conditions are being used to instantiate variables, which is incorrect for the semantics of the language, as (future & shadow) conditions should only be used to ensure future states are in their correct states. After letting Deep Seek know of this mistake, it decides to try and instantiate variables in a (future & shadow) section once again attempting different semantics. I once again informed it that (future & shadow) blocks should not be used for data instantiation. Finally, for Deep Seek’s 3rd attempt at creating a solution, Deep Seek tries to instantiate variables in a forall loop instead, which is also not correct. There is never a need to instantiate variables within the protocol language like Deep Seek is attempting to do.

A screenshot of a computer program

AI-generated content may be incorrect.

*Figure 8: 3-Bit Token Ring (all 3 attempts erroring in the same manner)*

With Deep Seek being unable to produce a solution by itself, I once again presented the solution off the protocon website. Deep Seek once again broke down the solution explaining it step-by-step and presents an example once again showing tit understands how the solution is correct, wrapping up by stating that the solution is correct.

**

*Figure 9: 3-Bit Token Ring with Deep Seek understanding the solution*

With Deep Seek understanding the solution, I then asked it to produce a scaled solution with a larger number of nodes. At this time, Deep Seek chose an N value of 5, and it produced a solution (LLMGiven/3-Bit/ScaledSolution.prot). Just like the previous solutions, Deep Seek walks through the solution after giving it to me trying to prove that the solution it generated is correct, and that it should not fail under any verification tests. Running this through protocon once again, it turns out Deep Seek is correct, and for the first time it produced a solution that verifies.

A screen shot of a computer

AI-generated content may be incorrect.

*Figure 10: 3-Bit Token Ring verification with a Scaled Solution*

With the first scaled solution finally working, I then asked Deep Seek to scale the solution up to value of N=20. Deep Seek once again produces a solution, which looks correct when looking over the code itself. After a long time of verification through Protocon, the tool also verifies that the scaled code to N=20 is also verifiable as being correct.

With Deep Seek being able to give some valid solutions for 3-Bit specification scaling, I asked once again to work on the 6-State token to present me with a valid solution. The first solution it gave (LLMGiven/6-State/Attempt4.prot) was unsuccessful, with an error in semantics around the shadow action. It is worth noting though, that now that it learned a bit with the 3-Bit this generation of a solution models more of what the 3-Bit solution looks like than the 6-State solution we original gave it. With this presented back to Deep Seek, it once again tried to generate solutions, which had errors due to not using the correct guard syntax (LLMGiven/6-State/Attempt5). The next attempt (6) caused a deadlock, and when pointed out to the LLM, it tries to add an invariant condition (future and shadow) to the solution, which causes the issue of the invariant not having the same style in all places. And after trying many different attempts to inform the LLM that using these invariant condition statements is not allowed, the best prompt that seemed to correct this issue was stating: “Invariant’s cannot be adjusted or declared within the (future & shadow) sections of the language, they are only used to ensure certain actions are happening”. With this finally fixed we end up with a deadlock on the current file (Attempt 10).

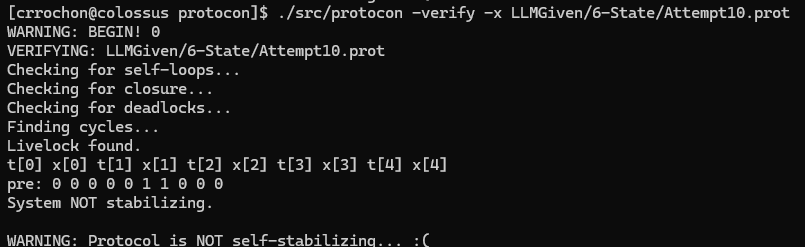
**

Figure 11: Image of Attempt10 Deadlock for 6-State System

When tracing this deadlock, working with Professor Ebnenasir, we were unable to resolve where the deadlock was occurring, and when talking with Alex Klinkhammer (someone who worked on developing Protocon), it turned out the issue was that all actions were being defined as shadow actions, when this is the actual actions we wanted to happen, not just imagine happening, so those actions needed to be changed to direct actions. Correcting this issue without the LLM as I was still learning about the Shadow actions, allowed a new livelock to occur and be traced (LLMGiven/6‑State/Livelock (Attempt10) Walkthrough.docx). This livelock trace was then shown to the LLM, which produced another Attempt (11), where another livelock occurred, which could also be traced. (Livelock (Attempt11) Walkthrough.docx). After tracing and showing the LLM this next livelock, and explaining this issue is where the guards are not cycling or changing, it is locked in a cycle, the LLM produced Attempt12, where it tries to use an if statement to get around the issue. This crashes the verification though, as if statements are not allowed in Protocon, and after informing the LLM of this issue, it finally produces Attempt13, where the system once again runs into an issue, this time a deadlock. With so many issues trying to get the LLM to build a 6-State correct attempt, we decided to move on to a different, and hopefully easier specification for it to learn from.

With this in mind, we moved to the Maximal Matching specification on One Ring. This specification is easier than the token ring passing, so the goal is to help the LLM learn more about Protocon specification, and have it improve its ability to create a solution. Feeding the specification and asking for an explanation as well proves an easy thing for the LLM to do, as it walks through each section of the specification and explains it in good detail

A screenshot of a computer

AI-generated content may be incorrect.

Figure 12: Image of the Maximal Ring Specification Description

The first attempt at a solution doesn’t follow the language semantics I have corrected before and starts out where it is trying to use if statements once again within the solution. Informing the LLM once again of the form that a specification should take (guard followed by actions with no if statements), this allows it to create the first attempt (LLMGiven/1‑Bit\_Match\_Ring/Attempt1.prot). This first attempt has an error around the puppet actions where it forgot a ‘;’, and when comparing it with the actual solution, it is close, yet it uses a whole different array of variables, where it uses shadow and puppet variables. When asking the LLM why it used shadow and puppet variables, it replies saying they should be added for the introduction of nondeterminism and the systems reactions.

A white background with black text

AI-generated content may be incorrect.

Figure 13: Image of LLM justifying variable usage

Once correcting the LLM’s mistake, the first deadlock appears. Once tracing through the deadlock, it is obvious that no guards evaluate to true for an iteration, which is causing this deadlock. After tracing the state (LLMGiven/1-Bit\_Match\_Ring/Deadlock (Attempt1) and feeding this to the LLM explaining that with the pre-state no guards are evaluating to true, the next attempt at a solution is given. This next attempt (2), results in a different error stating “The Invariant includes non-shadow variables.” This is because in the most recent attempt the LLM attempts to add more future and silent clauses, except it tried doing it with a puppet variable which is not allowed.

A black background with white text

AI-generated content may be incorrect.

Figure 14: Image of the (future & silent) clause causing the invariant error

To correct this issue, I first fed the LLM the error message, hoping that it would detect where it went wrong and correct its mistake, unfortunately it could not find it. Instead, the LLM suggested converting all the puppet variables into the shadow variables, instead of just removing the one check. After informing the LLM of the error on line 16, it let me know that I was correct and produced Attempt3. Attempt3, when plugged into Protocon, produces the same deadlock as Attempt1, and after some manual inspection it is obvious why: Because the puppet actions had no change. After informing the LLM that this system still deadlocks, and showing the deadlock trace once again, I also let it know that the puppet actions need to be adjusted to ensure one guard is always active. To try and correct this issue, the LLM adds two more actions to try and ensure that a puppet variable will always flip, yet this still results in a deadlock. Runing through this attempt produced a result where eventually all gates would deadlock on a situation (LLMGiven/1‑Bit\_Match\_Ring/Deadlock (Attempt4).docx). After feeding this result back to the LLM, it summarized the error with the current solution it generated, and Attempt5 was received.

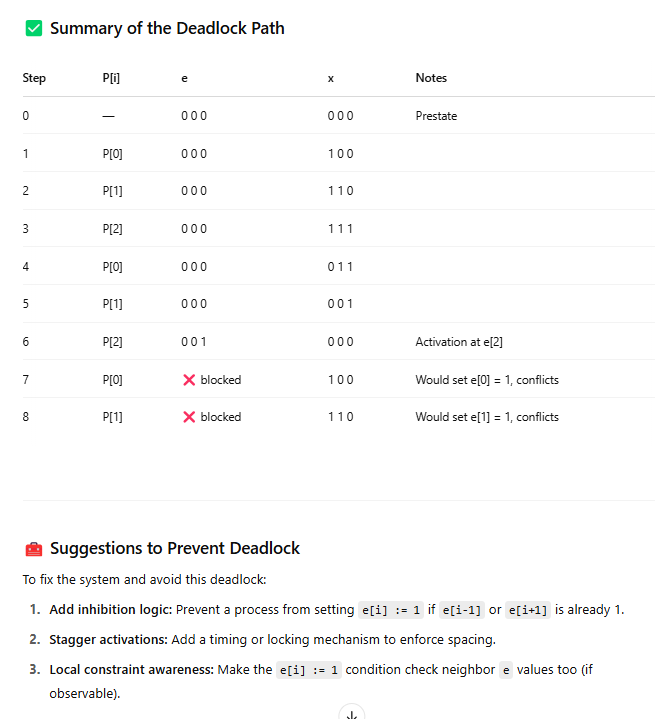


Figure 15: Image of the LLM explaining the deadlock after I gave it the trace

Running this new attempt in Protocon results in the same deadlock pre-state, with the major changes that the LLM did with this solution was add more checks within the guard to try and ensure it did not enter an invalid state.

One thing I wish I completed or attempted while working on this project, was to try and feed the LLM a specification without any shadow variables, as it seems the LLM struggles to understand shadow variables, and the fact they are there to define the system abstractly, but should not be used to define states when working with creating the solution. I wonder if this would help to produce better results that may have allowed for the LLM to learn better. The other thing I wish I could have completed was the 1-Bit Maximal Ring, to try and see if the LLM would stay in a deadlock feedback loop, or if it would have eventually produced the solution, although it appears very bleak that the LLM has the capability to do so at this time.