

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 Ebneenasir. 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 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.

All solutions used throughout this paper can be found at: [LLM Examles GitHub](#)

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.

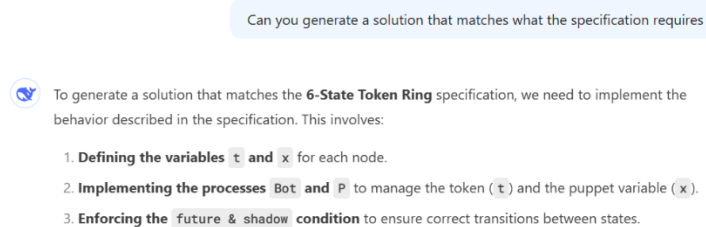


Figure 1: 6-State Token Ring Deep Seek Query to generate a solution

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.

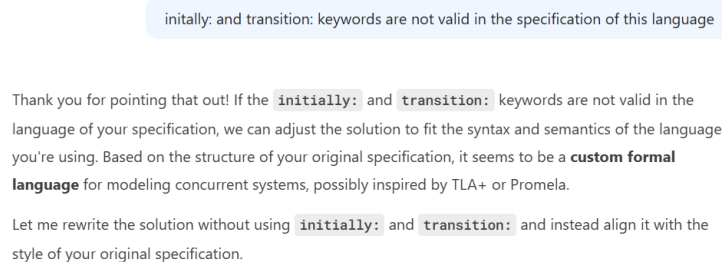


Figure 2: 6-State Token Ring Deep Seek Query for the 2nd solution attempt

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.

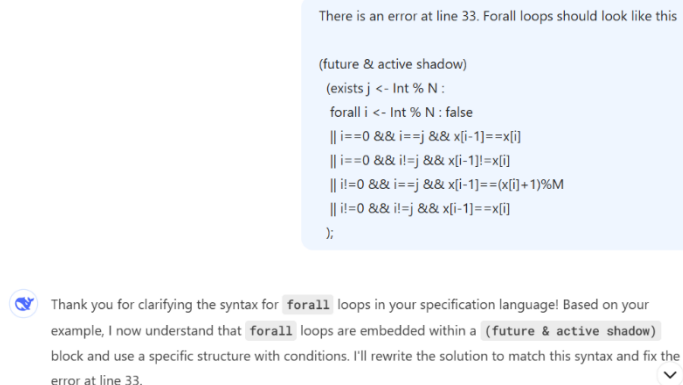


Figure 3: 6-State Token Ring Deep Seek Query for the 3rd solution attempt

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:

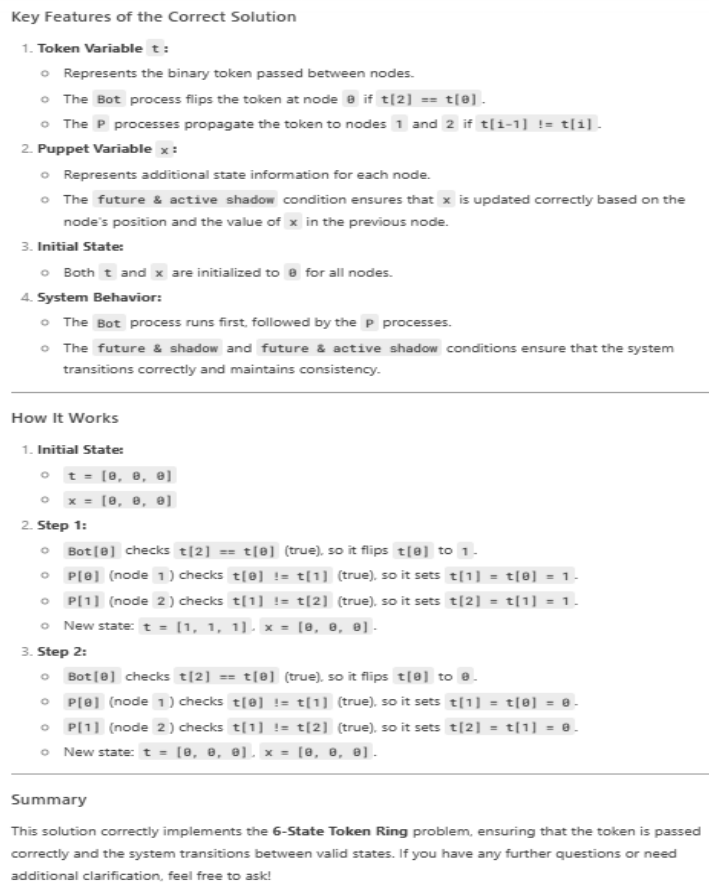


Figure 4: 6-State Token Ring Deep Seek analyzing the correct solution

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.

```
[crrachon@colossus protocon]$ ./src/protocon -verify -x LLMGiven/6-State/ScaledSolution.prot
WARNING: BEGIN! 0
VERIFYING: LLMGiven/6-State/ScaledSolution.prot
Checking for self-loops...
Checking for closure...
Checking for deadlocks...
Deadlock found.
t[0] x[0] t[1] x[1] t[2] x[2] t[3] x[3] t[4] x[4]
pre: 0 0 0 1 0 1 0 1 0 3
System NOT stabilizing.
WARNING: Protocol is NOT self-stabilizing... :(
```

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

Unfortunately, this first attempt when ran through Protocon (LLMGiven/6-State/ScaledSolution.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/ScaledSolution3.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.

```
[crrachon@colossus protocon]$ ./src/protocon -verify -x LLMGiven/6-State/ScaledSolution3.prot
WARNING: BEGIN! 0
VERIFYING: LLMGiven/6-State/ScaledSolution3.prot
Checking for self-loops...
Self-loop found.
t[0] x[0] t[1] x[1] t[2] x[2] t[3] x[3] t[4] x[4]
pre: 0 0 0 0 0 0 0 0 0 0
pre: 0 0 0 0 0 0 0 0 0 1
pre: 0 0 0 0 0 0 0 0 0 1
pre: 0 0 0 0 0 0 0 0 0 1
System NOT stabilizing.
WARNING: Protocol is NOT self-stabilizing... :(
```

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

The next protocol I tried to work with Deep Seek on was the 3-State 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.

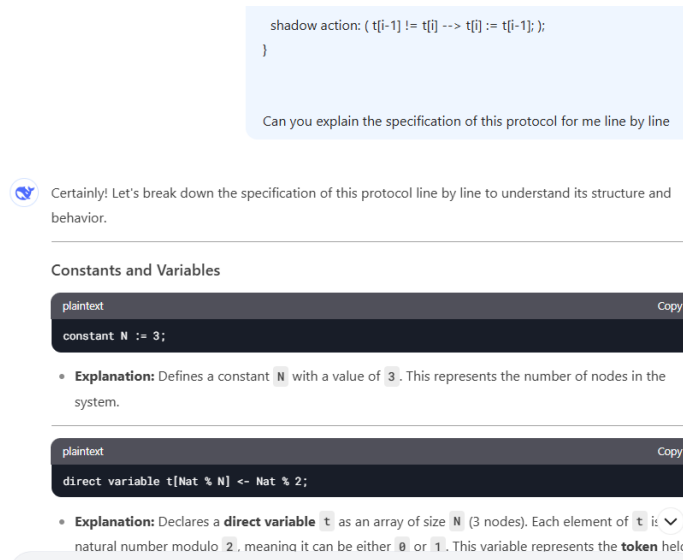


Figure 7: 3-State Token Ring specification understanding

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-State/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.

```
[crron@colossus protocon]$ ./src/protocon -verify -x LLMGiven/3-State/Attempt3.prot
Error at line 41 in text: forall i <- Nat % N :
  t[i] := 0; // Initialize token variable to 0
  e[i] := 0; // Initialize puppet variable e to 0
  ready[i] := 0; // Initialize ready variable to 0

// System behavior (future & shadow to ensure legitimate states)
(future & shadow)
(unique i <- Nat % N :
  i == 0 && t[i-1] == t[i] // Bot has the token
  ||
  i != 0 && t[i-1] != t[i] // P[i] has the token
);
Reason for error: Unknown.
FAILOUT: protocon tool
^^^ Host: colossus.it.mtu.edu
```

Figure 8: 3-State 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 it understands how the solution is correct, wrapping up by stating that the solution is correct.

Step 2:

- P[1] (node 1) checks $t[0] \neq t[1]$ (true), so it sets $t[1] := t[0] = 1$.
- P[2] (node 2) checks $t[1] \neq t[2]$ (true), so it sets $t[2] := t[1] = 1$.
- New state:

```
t = [1, 1, 1]
e = [0, 0, 0]
ready = [0, 0, 0]
```

Step 3:

- Bot (node 0) checks $t[2] == t[0]$ (true), so it flips $t[0]$ to 0.
- New state:

```
t = [0, 1, 1]
e = [0, 0, 0]
ready = [0, 0, 0]
```

This solution is correct and adheres to the formal specification language used in your system. Let me know if you need further clarification!

Figure 9: 3-State 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-State/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.

```
[crruchon@colossus protocon]$ ./src/protocon -verify -x LLMGiven/3-State/ScaledSolution.prot
WARNING: BEGIN! 0
VERIFYING: LLMGiven/3-State/ScaledSolution.prot
Checking for self-loops...
Checking for closure...
Checking for deadlocks...
Finding cycles...
Checking behavior within the invariant...
Checking for deadlocks in new invariant...
Max strong/weak async steps to converge: 18/7
System is stabilizing.
WARNING: Protocol is self-stabilizing!
```

Figure 10: 3-State 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 able to give some valid solutions for scaling, I asked it once again to work on the 6-State token to present me with a valid solution. The first solution it gave (LLMGiven/6-State/ScaledSolution4.prot) was unsuccessful, with an error in semantics around the shadow action. With this presented back to Deep Seek, it once again tried to generate a solution.