Formal Verification of Hardware and Software Systems Final Project: Verification of Cache Protocols using Model Checking

CSEE W6863 Fall 2022
Cam Coleman, Qiao Haung
cc4535, qh2234

1 Qiao's Side

1.1 Contributions

Our main contribution is model-checking 3 MSI cache protocol variants, which ranged from 6 to 14 states for the cache controllers and 4 states for the memory controller, under different bus assumptions. The difference between those variants is they work in different buses: The simplest one only works on the atomic-request-atomic-transaction bus, while the most complicated one works on the real-world non-atomic bus. What interests us is how the non-atomicity affects the design of the MSI protocol, so we modeled 3 different buses as well, and - Verified that these variants are correct under corresponding bus atomicity assumptions - Run the simpler protocols under non-atomic buses, verified that they went wrong, and used the error-tracing feature to see why they didn't work and how the more advanced variants fix the problem. In the following section, we will talk about which MSI variants we model checked, how we model them along with different buses with TLA+ and our results.

1.1.1 MSI cache coherence model

Consider one cache line (or other units for atomic data storage/transition) of data under the MSI protocol. For each core in the system, as its name suggested, the data has 3 stable states: "M/Modified" means the data is locally modified and not necessarily synced to the memory, "S/Shared" means that the data is in the read mode and potentially several cores have a copy of this data simultaneously, "I/Invalid" means that data is either not in the local cache or outdated. It's trivial to model the transitions if we only consider the stable states: - Before any operation, sync the local cache of other cores to the memory and make sure other cores are not in the M state - When core A writing data, we set A to the M state and other cores to the I state - When core A reading data, we set A to the S state

While the idea is straightforward, it's not as easy as it sounds to implement under concurrent circumstances. The way that MSI protocol sync between the cores is bus snooping: every core observe the request sent by other cores on the bus and process them in the same order. An example of a core read timeline would look like this: 1. Core A sends a request on the bus 2. Every core process the request (A included) 3. The memory sends back the requested data 4. Core A receives the data and updates its state

There are 3 kinds of requests that each core can send to the bus: - GetS: The core wants to read from the memory and become the "S" state - GetM: The core wants to write to the memory and become the "M" state. It's called "Get" because the core is not in the "M" state now, so it has to fetch the most recent value and make sure other cores have written their changes. - PutM: The core wants to write to the memory while it's in the "M" state. There is an Upgrade request we didn't consider because it's optional and doesn't introduce more insight. In our TLA+ program, these are all 3 requests that a core can send to the bus. Moreover, a core can also snoop and process a request on the bus, or write to the local data if permitted, in other words, in the state "M". We model the behavior of the cores in TLA+ as follows:

```
coreAction(core) ==
    \/ coreBusEvent(core) \* snoop and process the request on the bus
    \/ getS(core)
    \/ getM(core)
    \/ putM(core)
    \/ /\ write(core)
       /\ UNCHANGED << state, bus, queue >>
```

The `coreBusEvent(core)` function implements the state transition of the MSI protocol, transient states included.

The memory also has its state machine, but it can't send requests subjectively. All that the memory can do is process the requests on the bus and send/store data accordingly. Combined, we can model the MSI protocol in TLA+ as:

```
Next_MSI = \/ \E core \in Cores: coreAction(core)
           \/ memoryAction
```

So each step of the TLA+ model corresponds to a time slot on the bus where only one core is granted bus access. We choose this abstraction level to model the MSI protocol because we can both show the impact of bus characteristics on the variants of MSI protocols and limit the model complexity.

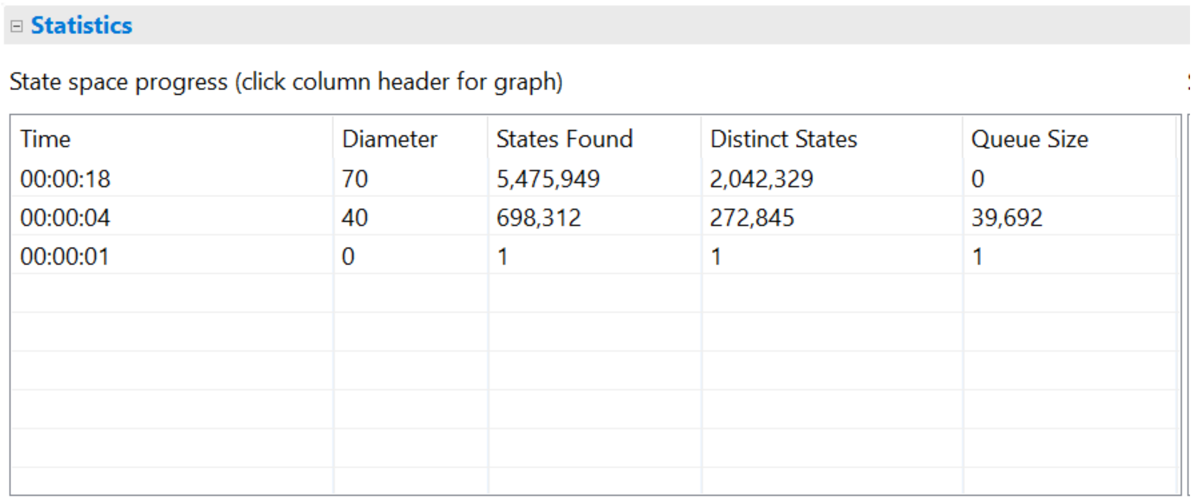
For a detailed description of the state machine for cores and memory, please refer to [1] and our code.

1.1.2 Bus model

We are interested in 2 characteristics of the bus: atomic-request, and atomic-transaction. Still, we only consider the requests about the same unit of data. The atomic request means that every request sent by the cores appears on the bus and gets processed immediately, in other words, there won't be any request between a core issuing a request and itself receiving it from the bus. The atomic transaction means that when core A issues a request, other requests will not appear on the bus until A receives the response for the request; However, other cores could issue requests during the transaction. An example of an atomic-transaction but not atomic-request bus could be: 1. Core A issues GetS 2. Core B issues GetS 3. A:GetS appears on the bus and every one process it 4. Memory put data on the bus and A receives the data from the bus 5. B:GetS appears on the bus and every one process it 6. Memory put data on the bus and B receives the data from the bus Since core B issues a GetS request when the first request from A has not finished, this bus is not atomic-request. However, in step 4 the first transaction for A:GetS finishes, then the request from B appears on the bus in step 5, so it has atomic transaction property.

For simplicity, we call a bus that is atomic-request and atomic-transaction as ARAT bus. Similarly, NRAT means not atomic-request but atomic-transaction, and NRNT means neither atomic-request nor atomic-transaction.

We used a queue to model ARAT and NRAT buses. Whenever a core or the memory sends requests or sends data, a request/data message is added to the global queue. The bus will repeatedly select a message from the queue and broadcast it to everyone. Without other constraints, this model is NRNT because the data returned by the memory could pend in the queue for a long time after the bus chooses other requests to broadcast, so we model the properties in TLA+ as:


Statistics

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
00:00:18	70	5,475,949	2,042,329	0
00:00:04	40	698,312	272,845	39,692
00:00:01	0	1	1	1

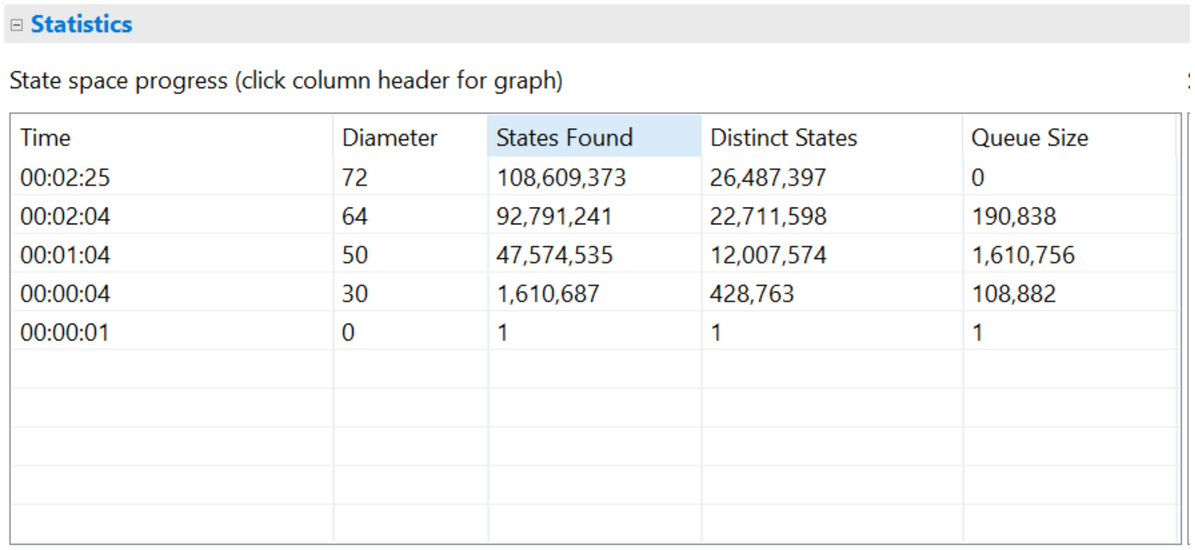
```
atomicRequest ==
    /\ Cardinality(queue') <= 1
    /\ isRequest(bus) => (\A msg \in queue': isData(msg))

atomicTransaction ==
    /\ Cardinality({msg \in queue': isData(msg)}) <= 1
    /\ (Cardinality(queue') < Cardinality(queue) /\ isRequest(bus)) =>
        \A msg \in queue': isRequest(msg)
```

where `bus` means the broadcasting message, and `queue` means the pending ones. What the code means is that if - there is never more than 1 pending message - no request can be sent to the queue when there is another request broadcasting then the bus is ARAT; If - there is never more than 1 pending data message - data messages get broadcasted before requests then the bus is NRAT. A bus that has ARAT/NRAT properties doesn't necessarily satisfy those TLA+ statements, but it's general enough and it creates 3 kinds of buses to model check our MSI variants. Moreover, one can hardly write a single statement equal to the ARAT/NRAT property without getting into trouble, and we'll discuss this issue later.

Our NRNT bus model is different from the model above. In short, we used 2 FIFO queues, one for request and one for data, to model the NRNT bus, because the protocol may deadlock if requests and data share one bus.

1.1.3 Property and results

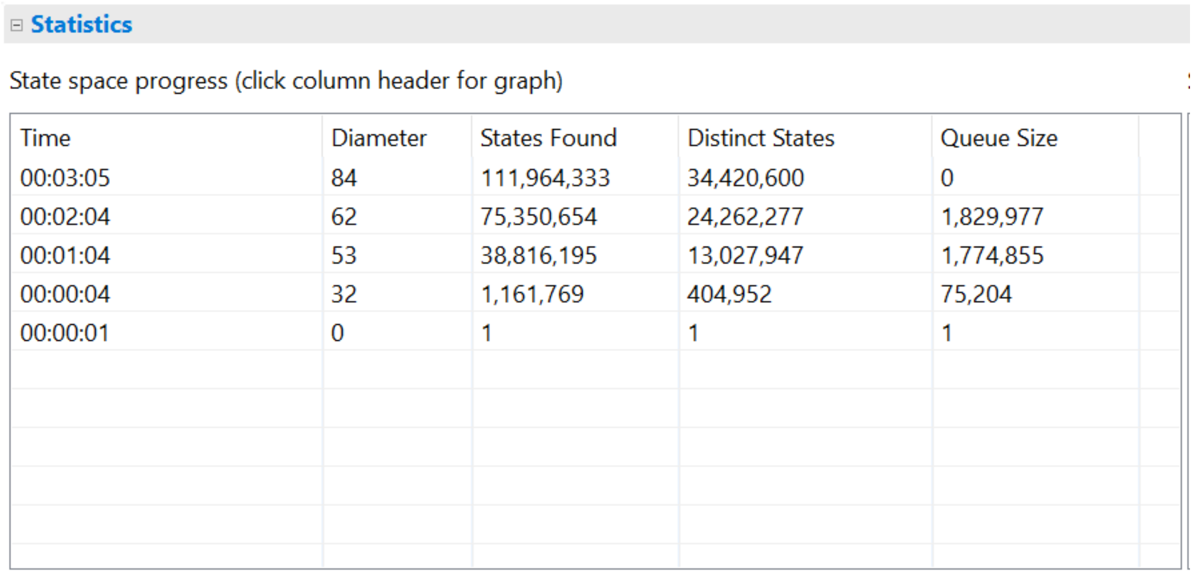
The property we want to check is simple: At any time the cache on all cores and the memory are coherent. The results show that all 3 variants of the MSI protocol passed the model checker under corresponding buses, but simpler protocol combined with non-atomic buses will yield counterexamples. The successful results are as follows:



Statistics

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
00:02:25	72	108,609,373	26,487,397	0
00:02:04	64	92,791,241	22,711,598	190,838
00:01:04	50	47,574,535	12,007,574	1,610,756
00:00:04	30	1,610,687	428,763	108,882
00:00:01	0	1	1	1



Statistics

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
00:03:05	84	111,964,333	34,420,600	0
00:02:04	62	75,350,654	24,262,277	1,829,977
00:01:04	53	38,816,195	13,027,947	1,774,855
00:00:04	32	1,161,769	404,952	75,204
00:00:01	0	1	1	1

1.2 Observations

1.2.1 PlusCal

Our first try was to model the protocol with PlusCal, which is a pseudocode-like language that can be translated into TLC+ code. The reason we try PlusCal first is that its syntax is close to programming languages, not math expressions, thus easy to learn and write. However, the expression power of PlusCal is lower than TLC+ and doesn't meet our requirements.

Initially, we modeled each core as a while loop process in PlusCal and used the `with` clause to select a core to run, but the compiler says that the `while` clause can't nest inside a `with` clause. As we must use `with` to select a core and `while` to model the repeating behavior of cores, we have to switch to plain TLA+. With a bit of investigation, we find that the issue is caused by the translation method from PlusCal to TLA+: The translator simulates the PlusCal program nearly on the machine code level with a pc counter. Within the `with` block, the pc counter can't change, but the while loop needs the pc counter to track. Anyway, it turns out that writing state machines with TLC+ is much easier than PlusCal because each transition can be nicely expressed in one statement.

1.2.2 Property formatting

It's natural to formulate the properties we want to check as something like `[]AtomicRequest => []Coherence`; However, we found it ending in deadlock in our setups. For example, some state machine transition is unreachable under ARAT buses, when we use the NRNT bus and the model check the temporal formula above with the ARAT MSI variant, the model checker will run into paths that don't satisfy ARAT, invoking unreachable transitions in the protocol and cause deadlock.

The problem occurs with `[]A => []B` kinds of statements, where `[]A` is the assumption of the model, in our case ARAT/NRAT. Generally, the model checker can't know if `[]A` holds or not until running the state machine to the end, which means it can't quit early and stop exploring the paths that `[]A` is false; If the behavior of the model is only defined for paths that `[]A` holds, the model checker will induce undefined behaviors. On the other side, the model has no reason to consider transitions outside its designed assumptions, thus forcing us to incorporate the model assumptions into the overall state machine, rather than sharing a bigger state space and checking for `[]A => []B` kinds of formulas.

Recall our discussion about writing a single statement equal to the ARAT/NRAT property. Even if we come up with such a statement `[]ARAT` that holds if and only if the bus has the property, checking for `[]ARAT => []Coherence` will end up in a deadlock. To avoid deadlock, we must incorporate the property into explicit state machine transition rules for the bus, which means the bus we describe is highly concrete and couldn't represent any ARAT buses. In conclusion, it's hard, if even possible, to check "this MSI protocol works under **any** ARAT buses" without specifying the undefined behaviors.

One possible approach is to suppose that if the model deadlock then ARAT doesn't hold, so we can still use a general bus model, check for `[]ARAT => []Coherence`, and discard those deadlock paths. But it has 2 problems too. First, the assumption itself is not obvious, thus requiring to prove or a derived model checking. Second, reaching deadlock is too late to realize ARAT doesn't hold, we much prevent deadlock beforehand. But preventing deadlock explicitly in the state machine in turn means that the bus model must be ARAT thus not general enough.

In the end, we used `Spec == Init /\ [][Next /\ atomicRequest]_vars` and model check for `[]Coherence`.

1.2.3 TLA+ as a language

Since all 3 variants share the basic MSI idea, we want to reuse code as much as possible and separate the protocol state machines with bus models if possible. However, we ended up copy paste a lot more than we intended to. The main reason is that TLA+ lacks polymorphism, which means we can't use an interface as a placeholder to model the high-level structure of the MSI protocol and fill the detailed implementation in variant-specific files.

The `UNCHANGED` clause of the TLA+ also impeded our code reuse effort. As we introduce more variables in a new variant, we must include it in every old function that doesn't change the new variable. If we write the `UNCHANGED` clause outside the function, it becomes the boilerplate code itself. We searched for the reason that the designer of TLA+ enforce the `UNCHANGED` clause, he said that by requiring explicitly listing unchanged variables, he could avoid many bugs about unintentionally changing a variable. However, some users suggest making this clause optional and identifying unchanged variables automatically.

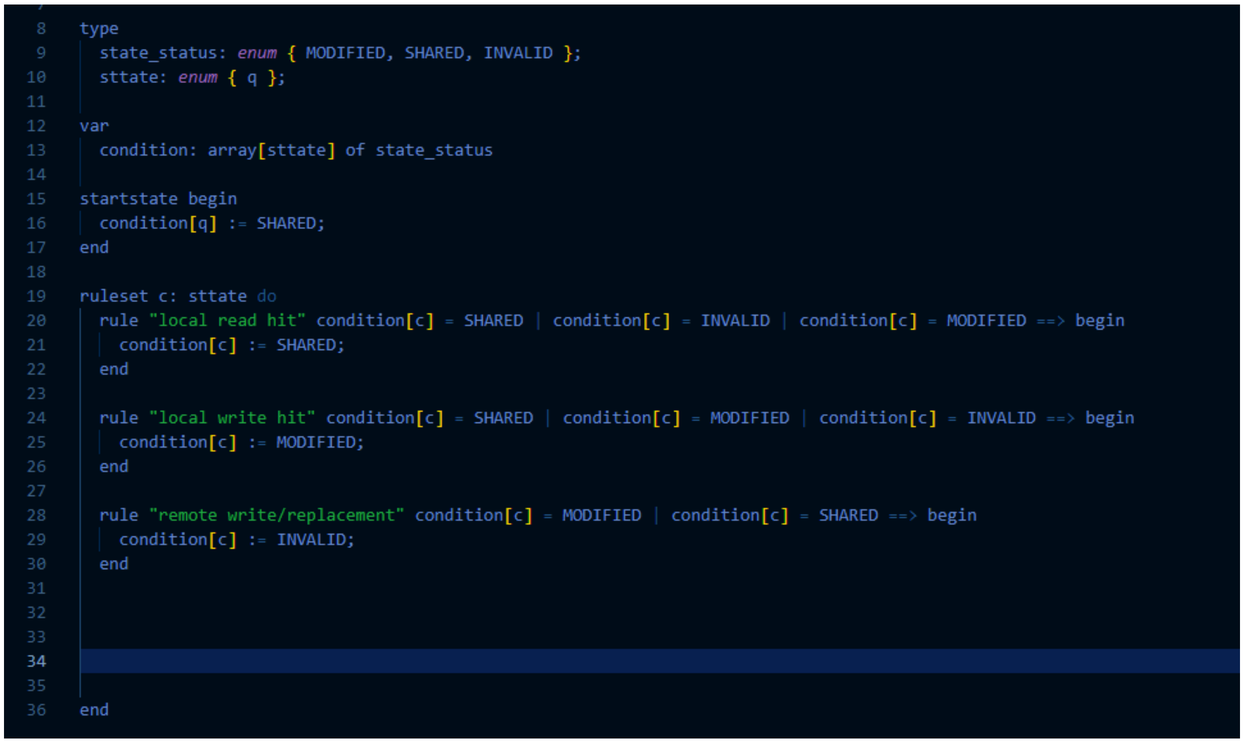
1.3 References

- [1] Sorin D J, Hill M D, Wood D A. A primer on memory consistency and cache coherence[J]. Synthesis lectures on computer architecture, 2011, 6(3): 1-212.

Cam's Side

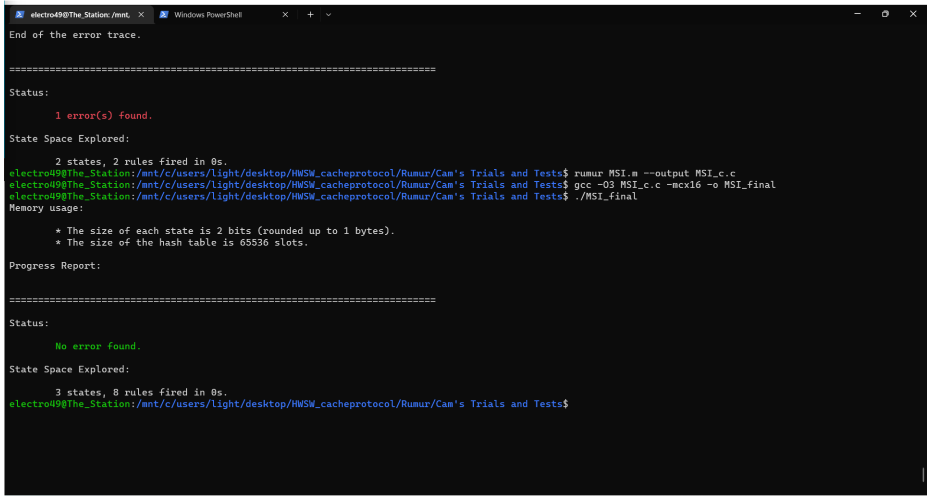
- **Timeline.** When referring to the first half and second half, the first half is before the Project Update, and the second half is after the Project Update.
  During the first half, I was not able to make much progress. I stopped attempting to implement Rumur and instead attempted to help out Qiao in TLA. During the first half I spent a little while learning more about TLA as well as its abilities to simulate formal verification of cache protocols, and while doing so I was attempting to find other means of formal verification of cache protocols using C/C++, which led me to attempt to use SPIN.
  I was using SPIN for the last week fo the first half. I was attempting to figure out how to utilize it and had a lot of trouble finding the software online, which is when Professor Ivancic suggested I use software similar to Ubuntu. After the conversation, I decided I should attempt to use and install Ubuntu as a last-minute resort toward using Rumur. I was able to successfully get Ubuntu installed and started using Rumur a couple of weeks into the second half, which was around the beginning of December.
  I spent the remaining 2 weeks learning about Murphi's language, how to use Murphi's language and Objective C with Rumur, and how to properly.

- **What was learned.**



```
type
  state_status: enum { MODIFIED, SHARED, INVALID };
  sttate: enum { q };

var
  condition: array[sttate] of state_status

startstate begin
  condition[q] := SHARED;
end

ruleset c: sttate do
  rule "local read hit" condition[c] = SHARED | condition[c] = INVALID | condition[c] = MODIFIED ==> begin
    condition[c] := SHARED;
  end

  rule "local write hit" condition[c] = SHARED | condition[c] = MODIFIED | condition[c] = INVALID ==> begin
    condition[c] := MODIFIED;
  end

  rule "remote write/replacement" condition[c] = MODIFIED | condition[c] = SHARED ==> begin
    condition[c] := INVALID;
  end




end
```

Figure 1: MSI protocol code, no error.



```
End of the error trace.


==========================================================================

Status:

        1 error(s) found.

State Space Explored:

        2 states, 2 rules fired in 0s.
electro49@The_Station:/mnt/c/users/light/desktop/HWSW_cacheprotocol/Rumur/Cam's Trials and Tests$ rumur MSI.m --output MSI_c.c
electro49@The_Station:/mnt/c/users/light/desktop/HWSW_cacheprotocol/Rumur/Cam's Trials and Tests$ gcc -O3 MSI_c.c -mcx16 -o MSI_final
electro49@The_Station:/mnt/c/users/light/desktop/HWSW_cacheprotocol/Rumur/Cam's Trials and Tests$ ./MSI_final
Memory usage:

        * The size of each state is 2 bits (rounded up to 1 bytes).
        * The size of the hash table is 65536 slots.

Progress Report:


==========================================================================

Status:

        No error found.

State Space Explored:

        3 states, 8 rules fired in 0s.
electro49@The_Station:/mnt/c/users/light/desktop/HWSW_cacheprotocol/Rumur/Cam's Trials and Tests$
```

Figure 2: What a no-error message looks like

- ***Rumur/Murphi.*** Murphi and Rumur are tools for formally verifying concurrent systems. They are designed to automatically check a model of a system for errors and other issues, such as deadlocks, livelocks, and unbounded behavior. Murphi and Rumur use a modeling language that is similar to a programming language, allowing designers to specify a system at a high level of abstraction. The tool then automatically synthesizes the high-level specification into a low-level hardware description, which can be used to implement the system on a specific hardware platform.
  Murphi and Rumur are particularly useful for verifying concurrent and hardware systems, such as distributed systems, communication protocols, and multi-threaded software. They are able to handle large and complex models efficiently, using techniques such as symbolic execution and counterexample-guided refinement to quickly and accurately check the model for errors and other issues. Murphi and Rumur are open source, which means that they are freely available to use and modify, and it has a wide range of applications in the field of formal verification.
  The reason why I compare Murphi and Rumur is due to timing. Murphi was released in 1992 but discontinued. A lot of software packages implemented and/or continued Murphi code that assisted in their formal verification of concurrent and hardware systems. They are nearly not as popular as Murphi was (except for SPIN), so there is rarely any documentation for both Murphi and Rumur.
- ***Using Rumur.*** When given an MSI protocol, you can use Rumur to simulate its correctness. Based on how the data is created, you can see step by step of the state machine process as well.
  Rumur is a program that is mainly used to provide feedback on the created state machine. It gives the option to provide extra printed information wihtin the program to track the progress of the state machine, as well as provide run speed and status. We see Rumur as a very basic program that is quck and and efficient at determining whether the state machine creates errors or not, the size of each states and its data usage, as well as states explored in an amount of time. Even with the heaviest of generated state machines run in under a second, but can often get to the double digits of bits in terms of the size of each state. Using the program, however, would require a lot of prior practice with Objective C as well as Murphi.

- **Encountered issues.** One of the biggest and main issues that happened with Rumur was the implementation process. Both Github and Ubuntu contained very old versions of Rumur, so it was extremely difficult to properly implement the software onto the computer. The reason why was because of the lack of resources that were found, especially in the first half.

  1. **Anaconda.** Anaconda properly installed on the Windows computer. The CMake, GCC, and GMP packages did not have the proper implemented functions in order to properly install Rumur. Due to channel complications and many other issues, Anaconda no longer works on Windows. There may be another attempt to reinstall Anaconda on Windows for a final attempt. Anaconda did not properly install on the Mac
  2. **Terminal/Powershell.** Windows was not able to properly install any of the packages needed to start the installation of Rumur in Powereshell. Mac properly installed all of

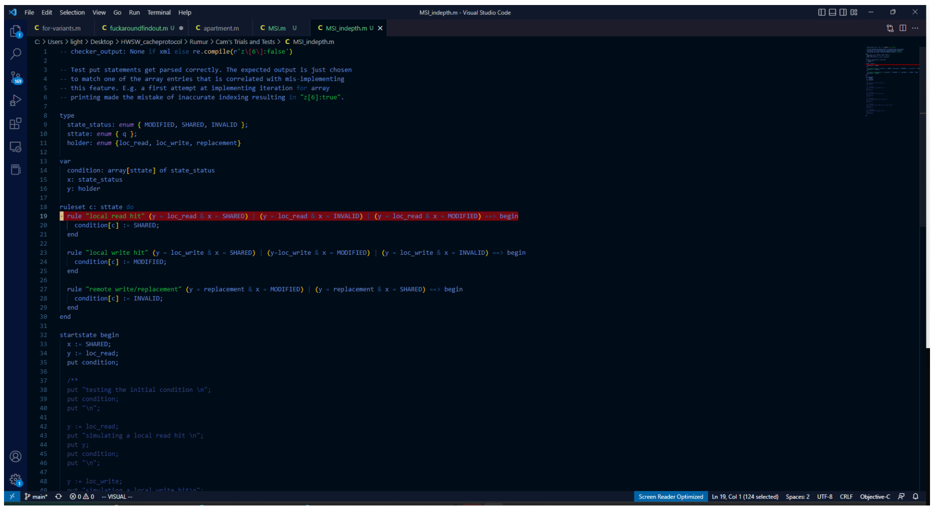
the packages using Homebrew, but there were still multiple issues found with the package installation when cmake was ran in the "build" directory. Cam may go back to that to see if the issue can be fixed.

3. **MobaXterm.** MobaXterm was not able to install the proper packages on both Mac and Windows due to some language conflictions as well as some packages bashing with each other. Cam will see if that problem can be fixed in their free time.
4. **Visual Studio Code.** VSCode was not able to be properly utilized because of a few issues. VSCode may not have been properly installed on Windows, which caused some issues with installing and using packages in VSCode's terminal. Cam may go back to try VSCode on the Mac.
5. **PuTTY.** Putty was not able to be utilized on the Windows because Cam was not able to find a proper environment to start doing work. May go back to see if Cam can do some research or ask questions to find a proper environment for PuTTY. Cam will do that before attempting to use it on the Mac.

During the second half, the only issue that was being encountered was the lack of resources. Like mentioned before, Rumur does not have a lot of supporting resources due to its lack of publishing and lack of usage. The only issue that was being encountered was attempting to work off of example files that were provided in order to create a new MSI protocol state machine.

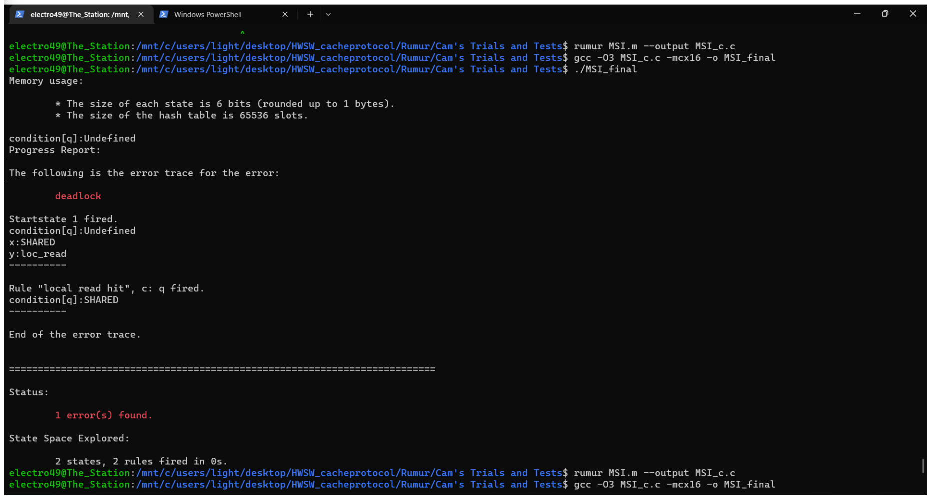
- **Unfinished work.** i would not say that there was no unfinished work. Just points in the project where I wish I had more time. I was successfully able to simulate an MSI program, and I was headed towards implementing an MSI program that used more states to see how much data it would use and how much time it would take. I unfortunately was not able to fit it into my schedule due to the amount of time that I spent trying to implement the software in the first place.

- **Limits and Advantages of using Rumur.**
  - Rumur is limited mainly because it was built very close to a software that was discontinued. A lot of similarities cause one to have to go back to find Murphi resources, which are already very hard to find.
  - Rumur is not an extremely visual tool. It is a simple tool which allows one to test for correctness as well as resources used for their state machine. Although it is a fast tool, it is not useful on a visual aspect, especially when using a language that is generally harder to understand.
  - Although Rumur is not the best to use visually, it is a great tool to use in order to understand how to create a state machine and how to use model checking. It also is decently descriptive in terms of errors that you may have, such as deadlocks and impossible rules set in place, as seen below. Deadlocks are often caused due to mutual exclusion or recourse ordering, and it is important not to have them. Deadlocks can be prevented thorugh careful design.



```
-- checker_output: None if xml else re.compile(r'z\[6\]:false')

-- Test put statements get parsed correctly. The expected output is just chosen
-- to match one of the array entries that is correlated with mis-implementing
-- this feature. E.g. a first attempt at implementing iteration for array
-- printing made the mistake of inaccurate indexing resulting in "z[6]:true".

type
  state_status: enum { MODIFIED, SHARED, INVALID };
  sttate: enum { q };
  holder: enum {loc_read, loc_write, replacement}

var
  condition: array[sttate] of state_status
  x: state_status
  y: holder

ruleset c: sttate do
  rule "local read hit" (y = loc_read & x = SHARED) | (y = loc_read & x = INVALID) | (y = loc_read & x = MODIFIED) ==> begin
    condition[c] := SHARED;
  end

  rule "local write hit" (y = loc_write & x = SHARED) | (y=loc_write & x = MODIFIED) | (y = loc_write & x = INVALID) ==> begin
    condition[c] := MODIFIED;
  end

  rule "remote write/replacement" (y = replacement & x = MODIFIED) | (y = replacement & x = SHARED) ==> begin
    condition[c] := INVALID;
  end
end

startstate begin
  x := SHARED;
  y := loc_read;
  put condition;

  /**
  put "testing the initial condition \n";
  put condition;
  put "\n";

  y := loc_read;
  put "simulating a local read hit \n";
  put y;
  put condition;
  put "\n";

  y := loc_write;
```

Figure 3: Code creating the deadlock problem



```
electro49@The_Station:/mnt/c/users/light/desktop/HWSW_cacheprotocol/Rumur/Cam's Trials and Tests$ rumur MSI.m --output MSI_c.c
electro49@The_Station:/mnt/c/users/light/desktop/HWSW_cacheprotocol/Rumur/Cam's Trials and Tests$ gcc -O3 MSI_c.c -mcx16 -o MSI_final
electro49@The_Station:/mnt/c/users/light/desktop/HWSW_cacheprotocol/Rumur/Cam's Trials and Tests$ ./MSI_final
Memory usage:

        * The size of each state is 6 bits (rounded up to 1 bytes).
        * The size of the hash table is 65536 slots.

condition[q]:Undefined
Progress Report:

The following is the error trace for the error:

        deadlock

Startstate 1 fired.
condition[q]:Undefined
x:SHARED
y:loc_read
----------

Rule "local read hit", c: q fired.
condition[q]:SHARED
----------

End of the error trace.


==========================================================================

Status:

        1 error(s) found.

State Space Explored:

        2 states, 2 rules fired in 0s.
electro49@The_Station:/mnt/c/users/light/desktop/HWSW_cacheprotocol/Rumur/Cam's Trials and Tests$ rumur MSI.m --output MSI_c.c
electro49@The_Station:/mnt/c/users/light/desktop/HWSW_cacheprotocol/Rumur/Cam's Trials and Tests$ gcc -O3 MSI_c.c -mcx16 -o MSI_final
```

Figure 4: Description of deadlock problem

  - It is able to cleanly implement MSI into its software. Using Modified, Invalid, and shared, it can be implemented in multiple different ways. The best thing about Rumur is that it is rather flexible and you can put in very original and unique code ideas and still be able to properly implement what you are looking for.
  - As listed, a major advantage of Rumur is its ability to use limited resources in both time and bits. It is as flexible as any other coding language as well. Although it is not visual, it is very easy to implement hardware designs, including cache protocols, into the software for simulation. It is a good tool to give a description of errors that may be within the code. Other than finding an error, it is not very descriptive besides that. It seems to be software that is simply used to determine the efficiency and correctness of a state machine.

- **Sources.**
  - https://github.com/Smattr/rumur
  - https://manpages.ubuntu.com/manpages/focal/man1/rumur.1.html
  - https://www.eecs.umich.edu/courses/eecs570/discussions/w22/murphi.html