README

1. What are you trying to model? Include a brief description that would give someone unfamiliar with the topic a basic understanding of your goal.
    - Our goal with this project is to model a system as it runs multiple programs. The problem with this execution is that these programs often need access to shared memory. Systems use mutexes as a locking system to prevent multiple programs from accessing the same piece of memory, but with their usage comes another issue. Situations could arise where it's impossible to make any forward progress since different programs are competing for the same mutex, or a program could try to unlock a mutex they never had any possession of in the first place. Both of these scenarios would cause huge problems for the system, so we've modeled a system that runs the necessary programs while avoiding these hazards. Alternatively, you could think of this model as one that identifies deadlocks in a system since we'd easily be able to notice those as well.
2. Give an overview of your model design choices, what checks or run statements you wrote, and what we should expect to see from an instance produced by Sterling. How should we look at and interpret an instance created by your spec?
    - We model the project as a state transition model. Each state consists of runnable code from a various program and a record of the locking state of the mutexes, along with the owner program.
    - There are two different *run* statements: TransitionStatesSafe and *TransitionStatesUnsafe*. The results they produce are the opposite. The *TransitionStatesSafe* will generate a set of programs with a specific execution order, such that it will not cause a deadlock. This, however, **does not** mean the program is free of deadlock, since there could be an alternative execution order that does cause the deadlock. Perhaps, a more useful run would be *TransitionStatesUnsafe* since this run generates a set of programs with a specific execution order, which will cause the deadlock to happen. With this knowledge, it tells us about code patterns which can cause the deadlock, so that we can all avoid using them in a program.
    - To make sense of the runtime result, we suggest setting the following fields to the "Show as Attribute"
        1. owner
        2. locks
        3. operation
        4. target
    - With the styling set as the mentioned, there are two mains groups to be concerned. First, the Codes -- each code represents a single lock/unlock instruction within a program. The nextLine relation simply connects one code to the other into a chain; this is the ordering of the code in a program, with the arrow pointing to the next line of code to run. There could be many chains of code, each chain is a single program.
    - Secondly, there are states. The state represents the runtime state of running all those programs. From one state to another, our state machine will pick one line of runnable code to run and apply the result onto *locks* field (the record of occupied mutexes). You

may wonder, where is the runnable code? Well, they are in *toRun* field in each state. *toRun* points to the first line of code in each program that has not been run. You will see that in a state transition, there would be one *toRun* which gets moved to the next code in *nextLine* -- that is the running line.

3. At a high level, what do each of your sigs and preds represent in the context of the model? Justify the purpose for their existence and how they fit together.
    - Our model uses states to represent different points in time as it runs the necessary programs. Each State holds information about the next state, code that hasn't been run yet, and the mutexes that currently have locks on them. The other notable sig in our model is the Code sig, which represents one line of code. From each line of code, we need information about what program it belongs to and what command it wants to perform. In our model, each line of code can only lock or unlock a mutex. The rest of the sigs, Command, Unlock, Lock, Mutex, and Program are used to provide more information for our model. Command, which encompasses Unlock and Lock, is used to signify what action a piece of code wants to carry out. Mutex is a means of identifying mutexes and finally, Program is used to associate pieces of code with different programs.
    - To link States and prevent illegal mutex commands we use a number of predicates. The first pred we have is the ValidStates pred that ensures that there are no states which disobey the rules of the model. For instance, we would not allow code from multiple programs to exist in the set of code to run, code could potentially run in the wrong order. The next pred we have is the canTransition pred, which we use to make sure every state transition behaves as we want them to. In our model, this means only one line of code from toRun should be replaced with its nextLine and the appropriate change should be noticed in the set of locks. The other preds we include specify the desired start and end state and how the states should transition from start to end. We have the initState pred to indicate that all code should be reachable from the start and there should be no locks to begin with. Following that we have two preds representing two desired end states. The first, finalWithoutDeadlock specifies that the system should have no code left to run and by that point there should be no locks either. The alternative is finalWithDeadlock which makes the final state one with a deadlock. Our last two predicates, TransitionStatesSafe and TransitionStatesUnsafe, use finalWithoutDeadlock and finalWithDeadlock to construct chains of States that either end without a deadlock or with a deadlock.

Overview of Sigs and their Fields

- **State** - The state of the system
    - **next** - stores zero or one State, points to next State in the chain
    - **toRun** - a set of Code, represents all lines of code that could be run between the current and the next state. No two lines of code in toRun should belong to the same program, otherwise code could be run out of order. In the initial state, toRun points to the head of the Code chain for each program.
    - **locks** - a set that maps Program to Mutex, all locks in the current state. A relation from ProgramX to MutexY signifies that ProgramX holds MutexY.

- **Mutex** - The Mutex sig has no fields, symbolizes a mutex
- **Program** - The Program sig also has no fields. It exists to signify ownership of a Mutex
- **Code** - The Code sig represents a line of code. For our model, we chose to disregard lines of code that did not lock/unlock any mutexes.
    - **owner** - the owner field stores one Program, representing which Program the line of Code belongs to. Note that each line of Code can only belong to one Program.
    - **target** - the target field stores one Mutex, the target is the Mutex that the code will apply its operation to
    - **operation** - the operation field stores one Command, either a Lock or an Unlock on a mutex
    - **nextLine** - the nextLine field stores zero or one Code, representing the next line of code in the Program. Code in a Program must be run sequentially, so this field lets us create a chain of Code..
- **Command** - An abstract sig encompassing both Unlock and Lock
- **Lock** - The command used to lock a mutex
- **Unlock** - The command used to unlock a mutex