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

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

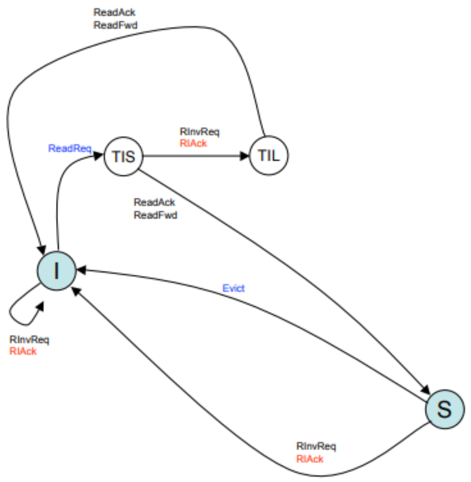

Figure 1: An example model of a cache protocol generated by Murphi (Remote Engine - I/S

Overview/Project Description

In this project we will attempt to verify a software application. We are currently debating on two ideas, so we have decided to propose a "plan B." Our main project will involve cache protocols using model checking, and the second will involve implementing SAT. The majority of this project proposal will be on plan A.

In this project, we will design and verify a cache coherence protocol on a multiprocessor system. Cache protocol can involve both software and hardware architecture, we will strive to include both but focus more on the software implementation. Cache coherence is a topic in computer architecture, where it is often used in multiprocessor systems. In a multiprocessor system, there can be multiple caches and multiple copies of those caches that contain different separate cache memory for each processor within the system; and within those multiple copies, the maintenance of the consistency raise a cache coherence problem. There are a wide range of cache coherence protocols, such as MSI Protocol and MOSI Protocol.

In the project we will verify a cache coherence protocol for a multiprocessor system using model checking. We will use a Murphi-like model checking framework, which will be talked about a little bit in the software application.

Software Application and Tools

In our project we plan on using a software in order to simulate the verification of cache protocols using model checking. We have found a model checker called Rumur, where it is a "formal verification tool for proving safety and security properties represented as state machines." It utilizes C programming and is required to be installed into our computers. Murphi is a model checking tool originally defined by a Stanford professor, and due to its discontinuation we found Rumur on Github, which is so far the only tool we have found that we will use.

Formal Verification Relation

Connection to Formal Verification is our goal to use model checking to solve our problems. Why Model Checking? Because it can help us see how we can possibly minimize the number of messages and hops per transition using F.V. It can also help us see if there is a way to break our system as well.

Project Goals

There are a couple goals. First goal is for us to find an optimal way to show cache protocol using Model Checking. Another is to see if we can find a way to break the system or create problems and see if we can find cache coherence problems using Model Checking to see if we can utilize model checking and formal verification to see if we can find alternatives to come up with a solid solution. Another outside of the project is to learn more about C as well as how we can implement usages of model checking within software and coding languages.

Plan B

There is a couple plan Bs that we have in mind

1. The first one will still involve cache protocols. We have also came up with the idea of using SPIN as well as Rumur. If we feel our project may be a little on the short end, we can use both Rumur and SPIN for different implementations and possible test run time and optimization between the two in order to extend our project

2. Our true plan B is implementation of SAT solvers. SAT solvers have played a major role in software and hardware verification, and it is used to find a depth first search (with backtracking to speed up the process) for Boolean operations in order to find optimal and correct paths. We would use Python in order to configure the SAT solver, where we would utilize DPLL algorithm and Boolean expressions in order to calculate using the SAT solver.

Algorithm 1: DPLL algorithm (implemented in `sat.py`)

```
Result: Either SAT or UNSAT
while not all variables assigned do
    if unit propagation returns conflict then
        return UNSAT
    end
    x ← choose splitting var
    val ← choose initial assignment for $x$
    create new decision level with $x$=val
    while unit propagation returns conflict do
        if decision level == 0 then
            return UNSAT
        end
        backtrack and set splitting variable for previous decision level to be the negation of
          the original choice
    end
end
return SAT
```

Figure 2: Psuedo code for the DPLL algorithm for SAT implementation