Jacquelyn Moreno
netID: jmore10
Collaborator: Melissa Kagaju

In this project my partner and I implemented the model checking and resolution inference methods. We demonstrated our implementations on modus ponens, the Wumpus World example, and the Horned Clauses example. For part 1, we implemented model checking. We used Professor Ferguson's code as a basis, and created a class ModelChecking which implements the Model interface. In this class we added methods for setting and getting values in a model, and for computing satisfiability of a model given a knowledge base and given a sentence. Then we created a class Prover which implements the algorithm in figure 7.10 of the textbook.

The model-checking algorithm, shown below as *checkAll( )*, takes in a knowledge base, the query alpha, a set of symbols, and a model. This algorithm is recursive, and in the first call the set of symbols contains all symbols in the knowledge base and the model has no true/false assignments to any variables. The KB and alpha always remain the same. In each call a symbol *temp* in the model is defined and removed from the list until there are no more symbols left to define. The function calls itself twice, once with *temp* set as true(*checkT*) and once with *temp* set as false(*checkF*). Once the list of symbols is empty this means that we have reached a base case and each symbol has been assigned a true/false value, so we now have a complete model. We then check if the KB holds within the model. If it doesn't, this means we cannot assume anything about alpha so we return true. If the KB does hold within the model then for alpha to be entailed, alpha must also hold in the model. If alpha holds, we return true otherwise we return false. The function returns the AND of each of the *checkT* and *checkF* calls, and the only time that one of these calls would return false is if the KB holds in a model and the query alpha does <u>not</u> hold in the model. This means that for a query alpha to be entailed by the KB, it must return true in each of the base cases.

```
public boolean checkAll(KB kb, Sentence alpha, List<Symbol> symbols, Model m) {
    if(symbols.isEmpty()) {
        if(m.satisfies(kb, m)) {
            return m.satisfies(alpha, m);
        }else {
            return true;
        }
    }else {
        Symbol temp = symbols.get(0);
        List<Symbol> rest = symbols.subList(1, symbols.size());
        boolean checkT = checkAll(kb, alpha, rest, m.set(temp, true, m));
        boolean checkF = checkAll(kb, alpha, rest, m.set(temp, false,m));
        return (checkT && checkF );
    }
}
```

We use this algorithm to compute the entailment of Modus Ponens, the Wumpus World example and the Horned Clauses example. The following figures show each of the models that are computed by the algorithm for Modus Ponens(right) and for the Horned Clauses(left) example. The model for Wumpus World was too large to show. The variables in the Horned Clauses are as follows: {Mag = magical, Mo = mortal, Ma = mammal, H = horned, My = mythical}. In our program we do not print these truth tables when displaying our answers and instead just print the last sentence which is the result of applying the algorithm.

Mag	Mo	Ma	H	My
true	true	true	true	true
true	true	true	true	false
true	true	true	false	true
true	true	true	false	false
true	true	false	true	true
true	true	false	true	false
true	true	false	false	true
true	true	false	false	false
true	false	true	true	true
true	false	true	true	false
true	false	true	false	true
true	false	true	false	false
true	false	false	true	true
true	false	false	true	false
true	false	false	false	true
true	false	false	false	false
false	true	true	true	true
false	true	true	true	false
false	true	true	false	true
false	true	true	false	false
false	true	false	true	true
false	true	false	true	false
false	true	false	false	true
false	true	false	false	false
false	false	true	true	true
false	false	true	true	false
false	false	true	false	true
false	false	true	false	false
false	false	false	true	true
false	false	false	true	false
false	false	false	false	true
false	false	false	false	false

KB entails (Unicorn is Horned): true

P	Q
true	true
true	false
false	true
false	false

KB entails Q: true

These truth tables help to show that the running time of model-checking is exponential in the number of propositional variables in the KB, because there are 2^n models where n is the number of variables. This is very inefficient because complex systems are likely to have many more variables. For part 1 my partner and I also completed two extra credit problems, problem 4: Liars and Truth Tellers, and problem 5: More Liars and Truth Tellers.

For part two of the project my partner and I chose to implement a resolution based theorem prover. To complete this we used Ferguson's code for CNF conversion, clauses and literals. We used these classes to implement the algorithm in figure 7.12 of the textbook. This algorithm operates under the fact that if a set of clauses is unsatisfiable, then by continually applying resolution to that set we will obtain the empty clause. The resolution algorithm is shown below as *checkResolution*(). This algorithm takes in a knowledge base and a query alpha. The algorithm first converts the knowledge base to a set of clauses in Conjunctive Normal Form and then adds the negation of alpha to the set of clauses. We add the negation of alpha because we are attempting a proof by contradiction, and by proving that the negation of alpha makes the set of clauses unsatisfiable, we prove that alpha *is* satisfiable. The algorithm then runs through every pair of clauses in the set and resolves them. If this resolution of clauses yields the empty clause, we return true because we have successfully proved that alpha is satisfiable. Otherwise, we keep resolving until eventually no new clauses can be added to the set. The *PL_Resolve()* method takes in two clauses and iterates through each pair of literals in the two clauses to determine if any of them are complementary. If they are complementary, those literals are removed from the clauses. If they are not, the clauses remain unchanged. After this step the clauses are combined into one clause and returned.

```
public boolean checkResolution(KB kb, Sentence alpha){
    Set <Clause> clauses = CNFConverter.convert(kb); //convert KB to CNF
    Set <Clause> negateA = CNFConverter.convert(new Negation(alpha)); // negate alpha and add it to clauses
    clauses.addAll(negateA);
    Set <Clause> newClauses = new HashSet<Clause>();
    while (true){
        for(Clause X : clauses) {
            for(Clause Y : clauses) {
                if(!X.equals(Y)) {
                    Set <Clause> resolvents;
                    resolvents = PL_Resolve(X, Y);
                    for(Clause Z: resolvents){
                        if (Z.isEmpty()){
                            return true;
                        }
                    }
                    newClauses.addAll(resolvents);
                }
            }
        }
        if (clauses.containsAll(newClauses)){
            return false;
        }
        clauses.addAll(newClauses);
    }
}
```

Below on the left is an example of the proof by resolution on the Horned clause query: Can we prove that the unicorn is horned? On the right is an example of the proof by resolution on Modus Ponens. In each of these examples, the empty clause is derived and so we conclude that the query is satisfiable. For clarity we have removed these print statements, so when you run our project you will just see the bottom line, which is the result of the proof by resolution. The runtime of proof by resolution is in some cases faster than model-checking, but if implemented incorrectly it has the potential to run infinitely, since larger and larger clauses can be generated. Thus, it is not always guaranteed to be faster than model checking, but in many cases it is more optimal since model checking has an exponential run-time.

```
Resolving {~My,~Mo} and {~Ma,H}
Returning :
{~My,~Mo,~Ma,H}

Resolving {~My,~Mo} and {Mo,H}
Returning :
{~My,H}

Resolving {~My,~Mo} and {~H}
Returning :
{~My,~Mo,~H}

Resolving {~My,~Mo} and {My,Ma}
Returning :
{~Mo,Ma}

Resolving {~My,~Mo} and {My,Mo}
Returning :
{}
Unicorn is horned:true
```

```
Resolving {~P,Q} and {~Q}
Returning :
{~P}

Resolving {~P,Q} and {P}
Returning :
{Q}

Resolving {~Q} and {~P}
Returning :
{~Q,~P}

Resolving {~Q} and {P}
Returning :
{~Q,P}

Resolving {~Q} and {Q}
Returning :
{}
KB entails Q: true
```