# 25 An Expert System Shell

**Chapter Objectives**

Completing the meta-interpreter for rule systems in Java
Full backtracking unification algorithm
- A goal-based reasoning shell
- An example rule system demonstration

The extended functionality for building expert systems
- Askable predicates
- Response to how and why queries
- Structure presented for addition of certainty factors

**Chapter Contents**



## 25.1 Introduction: Expert Systems

In Chapter 24, we developed a unification-based logic problem solver that solved queries through a depth-first, backward chaining search. In this chapter, we will extend those classes to implement two features commonly found in expert-system shells: the ability to attach confidence estimates, or certainty factors, to inferences (see Luger 2009 for more on certainty factors), and the ability to interact with the user during the reasoning process. Since all the classes in this chapter will extend classes from the unification problem solver, readers must be sure to have read that chapter before continuing.

In developing the expert system shell, we have two goals. The first is to explore the use of simple inheritance to extend an existing body of code. The second is to provide the reader with a start on more extensive modifications to the code that will be a valuable learning experience; the exercises will make several suggestions for such extensions.

**Certainty Factors**

The first extension to the reasoner will be to implement a simplified version of the certainty factor algebra described in Luger (2009). Certainty factors will be numbers between -1.0 and 1.0 that measure our confidence in an inference: -1.0 indicates the conclusion is false with maximum certainty, and 1.0 means it is true with maximum certainty. A certainty value of 0.0 indicates nothing is known about the assertion. Values between -1.0 and 1.0 indicate varying degrees of confidence.

Rules have an attached certainty factor, which indicates the certainty of their conclusion if all elements in the premise are known with complete certainty. Consider the following rule and corresponding certainty factor:

```
If p then q, CF = 0.5
```

This means that, if `p` is true with a confidence of 1.0 (maximum confidence), then `q` can be inferred to be true with a confidence of 0.5. This is the measure of the uncertainty introduced by the rule itself. If our confidence in `p` is less, than our confidence in `q` will be lowered accordingly.

In the case of the conjunction, or "and," of two expressions, we compute the certainty of the conjunction as the minimum of the certainty of the operands. Note that if we limit certainty values to 1.0 (true) and -1.0 (false), this reduces to the standard definition of "and." For the "or" operation, the certainty of the expressions is the maximum of the certainty of its individual operands. The "not" operator switches the sign of the certainty factor of its argument. These are also intuitive extensions of the boolean meaning of those operators.

Certainty factors propagate upward through the inference chain: given a rule, we unify the rule premises with matching subgoals. After inferring the certainties of the individual subgoals, we compute the certainty of the entire rule premise according to the operators for `and`, `or`, and `not`. Finally, we multiply the certainty of the premise by the certainty of the rule to compute the certainty of the rule conclusion.

Generally, certainty factor implementations will prune a line of reasoning if the certainty value falls below a certain value. A common pruning value is if the certainty is less than 0.2. This can eliminate many branches of the search space. We will not include this in the implementation of this chapter, but will leave it as an exercise.

## 25.2 Certainty Factors and the Unification Problem Solver

Our basic design strategy will be to make minimal changes to the representation of expressions, and to make most of our changes to the nodes of the solution tree. The reasoning behind this approach is the idea that the nodes of the solution tree define the inference strategy, whereas logical expressions simply are a statement about the world that is independent of its truth or reasoning. As a variation on truth-values, it follows that we should treat certainty calculations as a part of the system's inference strategy, implementing them as extensions to descendents of the class `AbstractSolutionNode`. This suggests we take `SimpleSentence` and basic operators to represent assertions independently of their certainty, and avoid changing them to support this new reasoning strategy.

The classes we will define will be in a new package called `expertSystemShell`. To make development of the expert system shell easier to follow, we will name classes in this package by adding the prefix "ES" to their ancestors in the package `unificationSolver` defined in the previous chapter.

**Adding Certainty Factors to Expressions**

We will support representation of certainty factors as an extension to the definition of `Rule` from the unification problem solver. We will define a new subclass of `Rule` to attach a certainty factor to the basic representation. We define `ESRule` as a straightforward extension of the Rule class by adding a private variable for certainty values, along with

standard accessors:

```
public class ESRule extends Rule
{
    private double certaintyFactor;
    public ESRule(ESSimpleSentence head,
             double certaintyFactor)
    {
         this(head, null, certaintyFactor);
    }
    public ESRule(ESSimpleSentence head, Goal body,
             double certaintyFactor)
    {
         super(head, body);
         this.certaintyFactor = certaintyFactor;
    }
    public double getCertaintyFactor()
    {
         return certaintyFactor;
    }
    protected void setCertaintyFactor(double value)
    {
         this.certaintyFactor = value;
    }
}
```

Note the two constructors, both of which include certainty factors in their arguments. The first constructor supports rules with conclusions only; since a fact is simply a rule without a premise, this allows us to add certainty factors to facts. The second constructor allows definition of full rules. An obvious extension to this definition would be to add checks to make sure certainty factors stay in the range -1.0 to 1.0, throwing an out of range exception if they are not in range. We leave this as an exercise.

This is essentially the only change we will make to our representation. Most of our changes will be to the solution nodes in the proof tree, since these define the reasoning strategy. To support this, we will define subclasses to both `SimpleSentence` and `And` to return the appropriate type of solution node, as required by the interface `Goal` (these are all defined in the preceding chapter). The new classes are:

```
public class ESSimpleSentence extends SimpleSentence
{
    public ESSimpleSentence(Constant functor,
             Unifiable... args)
    {
         super(functor, args);
    }
```

```
    public AbstractSolutionNode getSolver(RuleSet
            rules, SubstitutionSet parentSolution)
    {
        return new
            ESSimpleSentenceSolutionNode(this,
                (ESRuleSet)rules, parentSolution);
    }
}
public class ESAnd extends And
{
    public ESAnd(Goal... operands)
    {
        super(operands);
    }
    public ESAnd(ArrayList<Goal> operands)
    {
        super(operands);
    }
    public AbstractSolutionNode getSolver(RuleSet
        rules, SubstitutionSet parentSolution)
    {
        return new ESAndSolutionNode(this, rules,
            parentSolution);
    }
}
```

These are the only extensions we will make to the representation classes. Next, we will define reasoning with certainty factors in the classes `ESSimpleSentenceSolutionNode` and `ESAndSolutionNode`.

**Reasoning with Certainty Factors**

Because the certainty of an expression depends on the inferences that led to it, the certainty factors computed during reasoning will be held in solution nodes of the proof tree, rather than the expressions themselves. Thus, every solution node will define at least a goal, a set of variable substitutions needed to match the goal during reasoning, and the certainty of that conclusion. The first two of these were implemented in the previous chapter in the class `AbstractSolutionNode`, and its descendents. These classes located their reasoning in the method, `nextSolution()`, defined abstractly in `AbstractSolutionNode`.

Our strategy will be to use the definitions of `nextSolution()` from the classes `SimpleSentenceSolutionNode` and `AndSolutionNode` defined in the previous chapter. So, for example, the basic framework of `ESSimpleSentenceSolutionNode` is:

```
public class ESSimpleSentenceSolutionNode
    extends SimpleSentenceSolutionNode
    implements ESSolutionNode
{
    private double certainty = 0.0; //default value
    public ESSimpleSentenceSolutionNode(
         ESSimpleSentence goal, ESRuleSet rules,
         SubstitutionSet parentSolution)
    {
         super(goal, rules, parentSolution);
    }
    public synchronized SubstitutionSet
         nextSolution()
         throws CloneNotSupportedException
    {
         SubstitutionSet solution =
              super.nextSolution();
         // Compute certainty factor for the solution
         // (see below)
         return solution;
    }
    public double getCertainty()
    {
         return certainty;
    }
 }
```

This schema, which will be the same for the `ESAndSolutionNode`, defines `ESSimpleSentenceSolutionNode` as a subclass of the `SimpleSentenceSolutionNode`, adding a member variable for the certainty associated with the current goal and substitution set. When finding the next solution for the goal, it will call `nextSolution()` on the parent class, and then compute the associated certainty factor.

The justification for this approach is that the unification problem solver of chapter 24 will find all valid solutions (i.e. sets of variable substitutions) to a goal through unification search. Adding certainty factors does not lead to new substitution sets – it only adds further qualifications on our confidence in those inferences. Note that this does lead to questions concerning logical `not`: if the reasoner cannot find a set of substitutions that make a goal true under the unification problem solver, should it fail or succeed with a certainty of -1.0? For this chapter, we are avoiding such semantic questions, but encourage the reader to probe them further.

We complete the definition of `nextSolution()` as follows

```
public synchronized SubstitutionSet nextSolution()
      throws CloneNotSupportedException
{
    SubstitutionSet solution = super.nextSolution();
    if(solution == null)
    {
         certainty = 0.0;
         return null;
    }
    ESRule rule = (ESRule) getCurrentRule();
    ESSolutionNode child =
         (ESSolutionNode) getChild();
    if(child == null)
    {
         // the rule was a simple fact
         certainty = rule.getCertaintyFactor();
    }
    else
    {
         certainty = child.getCertainty() *
              rule.getCertaintyFactor();
    }
    return solution;
}
```

After calling `super.nextSolution()`, the method checks if the value returned is null, indicating no further solutions were found. If this is the case, it returns null to the parent class, indicating this branch of the search space is exhausted.

If there is a solution, the method gets the current rule which was used to solve the goal, and also gets the child node in the search space. If the child node is null, this indicates a leaf node, and the certainty factor is simply that of the associated rule. Otherwise, the method gets the certainty of the child and multiplies it by the rule's certainty factor. It saves the result in the member variable certainty.

Note that this method is synchronized. This is necessary to prevent a threaded implementation from interrupting the method between computing the solution substitution set, and the associated certainty, as this might cause an inconsistency.

The implementation of the class `ESAndSolutionNode` follows the same pattern, but computes the certainty factor of the node recursively: as the minimum of the certainty of the first operand (the head operand) and the certainty of the rest of the operands (the tail operands).

```
public class ESAndSolutionNode
    extends AndSolutionNode
    implements ESSolutionNode
{
    private double certainty = 0.0;
    public ESAndSolutionNode(ESAnd goal,
        RuleSet rules,
        SubstitutionSet parentSolution)
    {
        super(goal, rules, parentSolution);
    }
    public synchronized SubstitutionSet
        nextSolution()
        throws CloneNotSupportedException
    {
        SubstitutionSet solution =
            super.nextSolution();
        if(solution == null)
        {
            certainty = 0.0;
            return null;
        }
        ESSolutionNode head = (ESSolutionNode)
            getHeadSolutionNode();
        ESSolutionNode tail = (ESSolutionNode)
            getTailSolutionNode();
        if(tail == null)
            certainty = head.getCertainty();
        else
            certainty =
                Math.min(head.getCertainty(),
                         tail.getCertainty());
        return solution;
    }
    public double getCertainty()
    {
        return certainty;
    }
 }
```

This completes the extension of the unification solver to include certainty factors.

## 25.3 Adding User Interactions

Another feature common to expert system shells is the ability to ask users about the truth of subgoals as determined by the context of the reasoning. The basic approach to this is to allow certain expressions to be designated as askable. Following the patterns of the earlier sections of this chapter, we will define askables as an extension to an existing class.

Looking at the code defined above, an obvious choice for the base class of askable predicates is the `ESSimpleSentence` class. It makes sense to limit user queries to simple sentences, since asking for the truth of a complex operation would be confusing to users. However, our approach will define `Ask` as a subset of the `Rule` class. There are two reasons for this:

1. In order to query users for the truth of an expression, the system will need to access a user interface. Adding user interfaces to `ESSimpleSentences` not only complicates their definition, but also it complicates the architecture of the expert system shell by closely coupling the interface with knowledge representation classes.
2. So far, our architecture separates knowledge representation syntax from semantics, with syntax being defined in descendents of the `PCExpression` interface, and the semantics being defined in the nodes of the search tree. User queries are a form of inference (may the gods of logic forgive me), and will be handled by them.

As we will see shortly, defining `Ask` as an extension of the `Rule` class better supports these design constraints. Although `Rule` is part of representation, it is closely tied to reasoning algorithms in the solution nodes, and we have already used it to define certainty factors. Our basic scheme will be to modify `ESSimpleSentenceSolutionNode` as follows:

1. If a goal matches the head of a rule, it is true if the premise of the rule is true;
2. If a goal matches the head of a rule with no premise, then it is true;
3. If a goal matches the head of an askable rule, then ask the user if it is true.

Conditions 1 & 2 are already part of the definition of `ESSimpleSentenceSolutionNode`. The remainder of this section will focus on adding #3 to its definition.

Implementing this will require distinguishing if a rule is askable. We will do this by adding a boolean variable to the ESRule class:

```
public class ESRule extends Rule
{
    private double certaintyFactor;
    private boolean ask = false;
                    // constructors and certainty factor
                    // accessors as defined above
```

```
    public boolean ask()
    {
        return ask;
    }
    protected void setAsk(boolean value)
    {
        ask = value;
     }
}
```

This definition sets ask to false as a default. We define the subclass **Ask** as:

```
public class ESAsk extends ESRule
{
    public ESAsk(ESSimpleSentence head)
    {
        super(head, 0.0);
        setAsk(true);
    }
}
```

Note that **ESAsk** has a single constructor, which enforces the constraint that an askable assertion be a simple sentence.

The next step in adding askables to the expert system shell is to modify the method `nextSolution()` of `ESSimpleSentenceSolutionNode` to test for askable predicates and query the user for their certainty value. The new version of `nextSolution()` is:

```
public synchronized SubstitutionSet nextSolution()
  throws CloneNotSupportedException
{
    SubstitutionSet solution = super.nextSolution();
    if(solution == null)
    {
        certainty = 0.0;
        return null;
    }
    ESRule rule = (ESRule) getCurrentRule();
    if(rule.ask())
    {
        ESFrontEnd frontEnd =
            ((ESRuleSet)getRuleSet()).
                getFrontEnd();
        certainty = frontEnd.ask((ESSimpleSentence)
                        rule.getHead(), solution);
        return solution;
    }
```

```
        ESSolutionNode child =
             (ESSolutionNode) getChild();
        if(child == null)
        {
             certainty = rule.getCertaintyFactor();
        }
        else
        {
             certainty = child.getCertainty() *
                          rule.getCertaintyFactor();
        }
        return solution;
    }
```

We will define `ESFrontEnd` in an interface:

```
    public interface ESFrontEnd
    {
        public double ask(ESSimpleSentence goal,
                 SubstitutionSet subs);
    }
```

Finally, we will introduce a new class, `ESRuleSet`, to extend `RuleSet` to include an instance of `ESFrontEnd`:

```
    public class ESRuleSet extends RuleSet
    {
        private ESFrontEnd frontEnd = null;
        public ESRuleSet(ESFrontEnd frontEnd,
             ESRule... rules)
        {
             super((Rule[])rules);
             this.frontEnd = frontEnd;
        }
        public ESFrontEnd getFrontEnd()
        {
             return frontEnd;
        }
    }
```

This is only a partial implementation of user interactions for the expert system shell. We still need to add the ability for users to make a top-level query to the reasoner, and also the ability to handle "how" and "why" queries as discussed in (Luger 2009). We leave these as an exercise.

## 25.4 Design Discussion

Although the extension of the unification problem solver into a simple expert system shell is, for the most part, straightforward, there are a couple

interesting design questions. The first of these was our decision to, as much as possible, leave the definitions of descendents of `PCExpression` as unchanged as possible, and place most of the new material in extensions to the solution node classes. Our reason for doing this reflects a theoretical consideration.

Logic makes a theoretical distinction between syntax and semantics, between the definition of well-formed expressions and the way they are used in reasoning. Our decision to define the expert system almost entirely through changes to the solution node classes reflects this distinction. In making this decision, we are following a general design heuristic that we have found useful, particularly in AI implementations: insofar as possible, define the class structure of code to reflect the concepts in an underlying mathematical theory. Like most heuristics, the reasons for this are intuitive, and we leave further analysis to the exercises.

The second major design decision is somewhat more problematic. This is our decision to use the nextSolution method from the unification solver to perform the actual search, and compute certainty factors afterwards. The benefits of this are in not modifying code that has already been written and tested, which follows standard object-oriented programming practice.

However, in this case, the standard practice leads to certain cons that should be considered. One of these is that, once a solution is found, acquiring both the variable substitutions and certainty factor requires two separate methods: `nextSolution` and `getCertainty`. This is error prone, since the person using the class must insure that no state changes occur between these calls. One solution is to write a convenience function that bundles both values into a new class (say `ESSolution`) and returns them. A more aggressive approach would be to ignore the current version of nextSolution entirely, and to write a brand new version.

This is a very interesting design decision, and we encourage the reader to try alternative approaches and discuss their trade-offs in the exercises to this chapter.

## Exercises

1. Modify the definition of the `nextSolution` method of the classes `ESSimpleSolutionNode` and `ESAndSolutionNode` to fail a line of reasoning if the certainty factor falls below a certain value (0.2 or 0.3 are typical values). Instrument your code to count the number of nodes visited and test it both with and without pruning.

2. Add range checks to all methods and classes that allow certainty factors to be set, throwing an exception of the value is not in the range of -1.0 to 1.0. Either use Java's built-in `IllegalArgumentException` or an exception class of your own definition. Discuss the pros and cons of the approach you choose.

3. In designing the object model for the unification problem solver, we followed the standard AI practice of distinguishing between the representation of well-formed expressions (classes implementing the interface `unifiable`) and the definition of the inference strategy in the

nodes of the solution tree (descendents of `AbstractSolutionNode`). This chapter's expert system shell built on that distinction. More importantly, because we were not changing the basic inference strategy other than to add certainty estimates, we approached the expert system by defining subclasses to `SimpleSentenceSolutionNode` and `AndSolutionNode`, and reusing the existing `nextSolution` method. If, however, we were changing the search strategy drastically, or for other reasons discussed in 25.4, it might have been more efficient to retain only the representation and rewrite the inference strategy completely. As an experiment to explore this option, rewrite the expert system shell without using `AbstractSolutionNode` or any of its descendants. This will give you a clean slate for implementing reasoning strategies. Although this does not make use of previously implemented code, it may allow making the solution simpler, easier to use, and more efficient. Implement an alternative solution, and discuss the trade-offs between this approach and that taken in the chapter.

4. Full implementations of certainty factors also allow the combination of certainty factors when multiple rules lead to the same goal. I.e., if the goal `g` with subsitituions `s` is supported by multiple lines of reasoning, what is its certainty? (Luger 2009) discusses how to compute these values. Implement this approach.

5. A feature common to expert systems that was not implemented in this chapter is the ability to provide explanations of reasoning through How and Why queries. As explained in (Luger 2009), How queries explain a fact by displaying the proof tree that led to it. Why queries explain why a question was asked by displaying the rule that is the current context of the question. Implement How and Why queries in the expert system shell, and support them through a user-friendly front end. This front-end should also allow users to enter queries, inspect rule sets, etc. It should also support askable predicates as discussed in the next exercise.

6. Build a front-end to support user interaction around askable predicates. In particular, it should keep track of answers that have been received, and avoid asking the same question twice. This means it should keep track of both expressions and substitutions that have been asked. An additional feature would be to support asking users for actual substitution values, and adding them to the substitution set.

7. Revisit the design decision to, so far as possible, locate our changes in the solution node classes, rather than descendants of `PCExpression`. In particular, comment on our heuristic of organizing code to reflect the structures implied by logical theory. Did this heuristic of following the structure of theory work well in our implementation? Why? Do you believe this heuristic to be generalizable beyond logic? Once again, why?