A KBS Application:
Building a Wholesale Distributor

Suppose that you have been hired to build a decision-support system (DSS) for a large corporation that does business throughout the world. This corporation often chooses new wholesaler distributors for its products in other countries.

Sometimes the corporation would like to acquire an existing distributor.

The decision is, “Do we buy distributor X?”

Sometimes it intends to initiate a joint venture with an existing distributor

The decision is, “Do we work with distributor X?”

Why might the corporation need a DSS?

Reasons to Build a Program

The company employs a number of financial analysts who make recommendations on these decisions. Why write a program?

Financial analysts embody knowledge.

- The company would like to codify its policies so that it can record, examine, and change them.
- The company would like to ensure that the decisions are made consistently over time.

Financial analysts are a scarce commodity.

- The company would like to make decisions when analysts are busy with other tasks.
- The company would like to make decisions when analysts are not present.

Hence, the need for a decision-support system.

Problem Analysis

All wholesale distributor decisions depend on a number of factors:

competence	size of workforce
intensity	complementary products
objectives	compensation level
sales volume	competitive products
strategy	financial strength
geographic range	

These are the **parameters** of the decision.

The system will examine the data of the target corporation and return an answer of ‘yes’, ‘no’, or maybe a less certain answer such as ‘probably yes’, ‘probably no’, or ‘uncertain’.

These are the possible *decisions*.

Recognition Tasks

This is a **recognition** task. The system must determine if a particular label applies to a given situation. People have to make these decisions in all domains, in all sorts of contexts.

We could implement a DSS for this problem as:

- a search agent
- a logical inference agent
- a learning agent
- a planner
- ... or using a slew of other AI techniques.

Each technique will introduce some thorny implementation decisions...

Builders of DSSs have to implement recognition programs in all sorts of domains and all sorts of contexts.

Do we want to build solutions from scratch each time, relearning the implementation lessons that other programmers have learned?

A Common Design for the System

Write a rule base that associates patterns of parameter values with particular decisions:

IF	competence = high	**and**
	size of workforce = (> 50,000)	**and**
	intensity = low	**and**
	complementary products = none	**and**
	…	
THEN	decision = yes	(or no, or ...)

Each rule associates one pattern with one choice.

Making the decision involves finding the rule whose pattern (best) matches the input data and returning its choice.

Such a rule is called a **simple matcher**.
A system made of such rules does **simple matching**.

Simple Matching

Simple matching is a technique that is straightforward to implement as a logical inference agent, as long as the logic allows simple variables.

Many logical inference programs evolve into a set of simple matchers.

But simple matchers also have a number of significant drawbacks:

1. It is computationally intractable, even for relatively small problems.

2. It has no way to represent intermediate abstractions that may play a role in the decision.

3. A simple matcher is brittle in the face of uncertain or missing data.

The Problem...

How do you organize a system that makes choices from a small, discrete set of alternatives in a way that is reasonably efficient and easily modified?

Any solution to this problem should find an equilibrium among these competing *forces* :

- Making a decision should be computationally tractable in the face of many parameters and possible values.

- The system should be able to explain its decision in terms of relevant factors, not just in terms of all factors together.

- The system should be able to capture interactions among parameters but not be brittle in the boundary conditions.

…

… and The Solution

Any solution to this problem should find an equilibrium among these competing *forces* :

…

- Systems evolve over time. The mechanism should allow programmers to modify the patterns used to make decisions, add new patterns, etc., easily.

- The representation of decision-making patterns should not be so far from how humans make the decision that acquiring knowledge from domain experts becomes problematic.

Decompose the decision into a hierarchy of sub-decisions. Group parameters according to the sub-decisions that they affect. For each sub-decision, construct a simple matcher that maps the values of its inputs—either input data or the decisions of other simple matchers—onto a value for its decision. The result is a *Structured Matcher*.

Applying the Solution

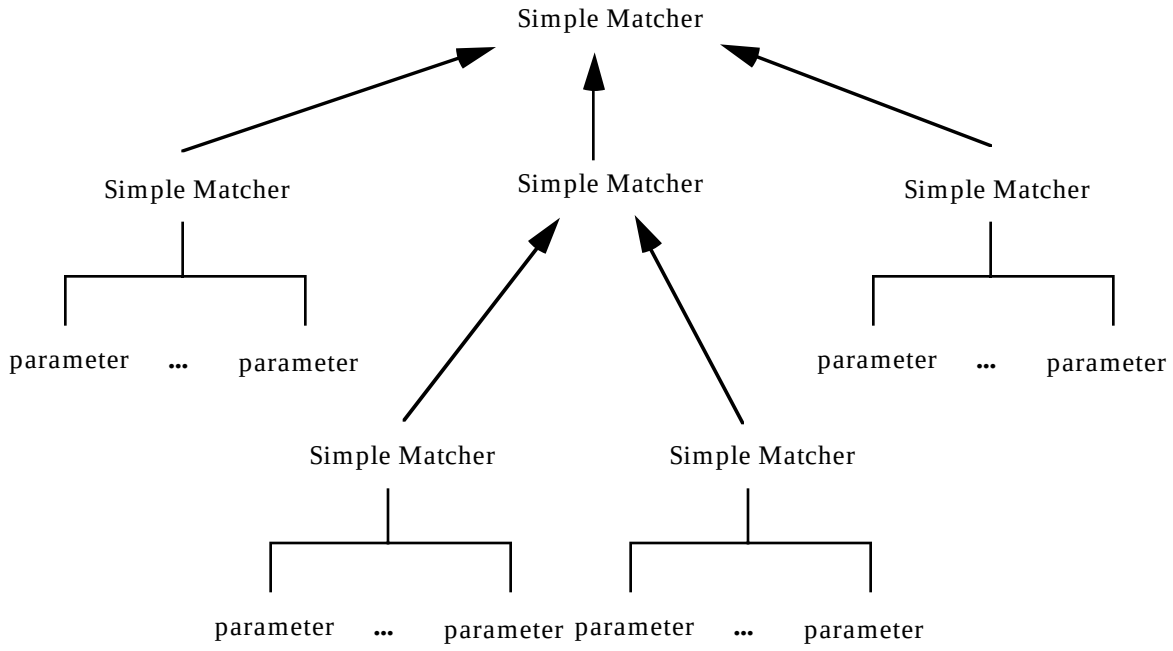

Figure 2. The Structure of a Structured Matcher

How can we apply this idea to the wholesale distributor selector system?

1. Identify meaningful sub-decisions.
2. Implement each simple matcher.
   - A decision tree.
   - A linear combination rule.
   - A set of pattern-matching rules.

A Example from the Wholesale Distributor Case

```
/* Rule set three: Determining Size Factors */

rule-3-1:
     if (sales-volume = strong or
        sales-volume = average) and
        financial-strength = strong
     then size-factors = 15.

rule-3-2:
     if (sales-volume = strong or
        sales-volume = average) and
        (financial-strength = average or
        financial-strength = weak)
     then size-factors = 10.

rule-3-3:
     if sales-volume = weak and
        (financial-strength = strong or
        financial-strength = average)
     then size-factors = 5.

rule-3-4:
     if sales-volume = weak and
        financial-strength = weak
     then size-factors = 0.
```

A Nice Benefit: Data Abstraction

Some rules deal with **observable data**, while others deal with values computed by other rules, such as “How big should our wholesale distributor in Mexico be?”

We will have several rules, each of which provides a particular answer. For example:

```
/* Rule set one: Final decision (recommendation) */
rule-1-1:
     if size-factors = A and
        sales-and-tech-personnel-factors = B and
        marketing-plan-factors = C and
        product-line-factors = D and
        market-coverage-factors = E and
        A+B+C+D+E >= 50
     then recommendation = accept.

rule-1-2:
     if size-factors = A and
        sales-and-tech-personnel-factors = B and
        marketing-plan-factors = C and
        product-line-factors = D and
        market-coverage-factors = E and
        32 <= A+B+C+D+E < 50
     then recommendation = consider.
```

```
rule-1-3:
     if size-factors = A and
        sales-and-tech-personnel-factors = B and
        marketing-plan-factors = C and
        product-line-factors = D and
        market-coverage-factors = E and
        A+B+C+D+E< 32
     then recommendation = reject.
```

A Nice Benefit: Organization

Making a decision such as “How big should our wholesale distributor in Mexico be?” will require several rules, each of which provides a particular answer.

```
/* Rule set three: Determining Size Factors (size-factors) */
               sales-volume     financial-strength   |    answer
rule-3-1:      >= average       = strong             |    15
rule-3-2:      >= average       <= average           |    10
rule-3-3:      = weak           >= average           |    5
rule-3-4:      = weak           = weak               |    0
```

A Nice Benefit: Organization

Or:

```
/* Rule set one: Final decision (recommendation) */
Let total-score = size-factors +
                  sales-and-tech-personnel-factors +
                  marketing-plan-factors +
                  product-line-factors +
                  market-coverage-factors

                 total-score      |    answer
rule-1-1:        >= 50            |    accept
rule-1-2:        < 50 and >= 32   |    consider
rule-1-3:        < 32             |    reject
...
```

The representation of the rules and *their relationship with one another* is **explicit**. This moves the creation of the rule base to a higher level (identifying and relating problem features) and makes all of the programming tasks easier to do.

A Requirement: Tractability

A simple matcher-based solution faces this computational task:

25 parameters
5 possible values for each

5^{25} = 298,023,223,876,953,125 different rules

A structured matcher-based solution faces this computational task:

25 parameters
6 structured matchers:
    1 at root and 5 intermediate decisions
5 possible values for each parameter and matcher

$(5 * 5^5) + 5^5 = 6 * 3125 = 18{,}750$ different rules

This example demonstrates how quickly divide-and-conquer approaches can improve the computational costs of solving a given problem.

The Programming Process in SM

To build a structured matcher, you must:

1. Design a hierarchy of problem features, or sub-decisions.
2. Group parameters according to the sub-decisions that they affect.
3. For each sub-decision, write a set of rules that determines the value of the decision based on the relevant parameters.

This provides significantly more guidance than:

1. Write a set of rules that determines the value of the decision based on the parameters.

But at what cost?

And what haven't we considered yet?

- How does our system handle uncertainty?
- How does our system explain its answer?
- Do we really want to write 18,000+ rules?

So...