STRUCTURED MATCHER — Object Behavioral

Author

Eugene Wallingford
Department of Computer Science
University of Northern Iowa
Cedar Falls, Iowa 50614-0507
wallingf@cs.uni.edu

Also Known As

Hypothesis Matching, Signature Tables, Candidate Selection

Intent

A Structured Matcher is useful when making choices from a small, discrete set of alternatives. It decomposes a complex decision into simpler decisions about relevant factors and then uses decisions about these factors to make the decision.

Example

Consider a decision support system for capital asset acquisitions (Wallingford and Sticklen 1985). The system helps users decide whether to purchase an asset or lease it from another company. Each of these alternatives leads to more specific decisions, such as whether to structure a leasing arrangement as a direct lease, a leveraged lease, or a conditional sale. All of these decisions depend on a number of different factors: the company's tax rate, financing costs, financial reporting requirements, characteristics of the market, and so on. For example, deciding whether to pursue a leveraged lease depends on ten factors:

- Availability of lease
- Existence of a potential lessor
- Business form of a potential lessor
- Lessor's risk aversion
- Nature of lessor's creditors
- Time window of opportunity
- Interest rate trends
- Access to debt/equity markets
- Sufficient working capital
- Existence of an independent trustee

Decisions such as these occur in every domain. In the capital asset acquisition scenario, the system performs a *recognition* task: it determines whether a particular label applies in a given situation. The choices facing the system are confidence values for the assessment, and the parameters of the decision are the expected features for situations that fit the category. In other cases, the system

performs a *classification* task, in which the choices are the categories themselves and the parameters of the decision are characteristics that distinguish one category from another.

Context

You are building a system in which many different kinds of parameters play a role in the decisions it makes.

Many software systems need to make decisions. Often these decisions involve the selection of one choice from a small set of alternatives, and several different kinds of parameters affect the decision. Many simple solutions to this kind of problem expose the underlying trade-offs that must be made in practice.

One way to make decisions of this sort is to directly associate patterns of parameter values with alternatives. For example:

If parameter 1 has value 1 and
parameter 2 has value 2 and ...
parameter *n* has value *n*,
then choose alternative *I*.

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. A system that uses this technique is called a *simple matcher*.

Simple matching suffers from at least three serious drawbacks. First, it is computationally intractable, even for relatively small problems. In the capital asset acquisition scenario, ten parameters play a role in deciding whether to structure a lease as a leveraged lease. Each of these parameters can take five different values. In the worst case, a simple matcher for the leveraged lease decision would require 5^{10} pattern-matching rules.

Second, it has no way to represent intermediate abstractions that may play a role in the decision. For example, one ordinarily would not make the leveraged lease decision in terms of the primitive input parameters. Instead, one would consider four higher-level factors: the feasibility of a leveraged lease, conditions that exist on the lessor, environmental factors, and financial factors. The input parameters are used directly in making these intermediate decisions, which are in turn used to make the leveraged lease decision.

Third, a simple matcher is brittle in the face of uncertain or missing data. If a value for one of the parameters is unavailable at run-time, any rule that matches on the parameter will fail, even if all the rest of its parameters match perfectly. Similarly, if a value for one of the parameters is incorrect in any way, the matcher will not be able to match the correct rule, even if all other data match

perfectly. So a simple matcher does not work well in environments with noise in the data.

One solution aimed specifically at the problem of brittleness computes decisions using a weighted sum of parameter values. Such an approach makes the system less dependent on the value of any particular parameter, but at two costs. First, the decision now resides in a combination function and threshold value, which makes explaining the decision in terms of intermediate decisions difficult. Second, and more critical, weighted sums cannot easily represent interactions among parameters. Most real problems include situations in which one parameter's value overrides another, and use of a weighted sum makes it difficult to capture this relationship.

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 bring the following *forces* into equilibrium:

- 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.
- Systems evolve over time. The decision-making mechanism should facilitate modifications to decision-making patterns.
- 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.

Solution

Decompose the decision into a hierarchy of sub-decisions using the Composite pattern (Gamma et al. 1995). 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 root of the hierarchy makes the primary decision. Interior matchers combine the decisions of matchers lower in the hierarchy. Matchers at the leaves of the hierarchy refer only to input data. The result is a *Structured Matcher*.

» In the capital asset acquisition scenario, decompose the decision of whether to seek a leveraged lease into four sub-decisions: feasibility, conditions on the lessor, general conditions, and financial factors. The parameters *Interest Rate Trends*, *Access to Debt/Equity Markets*, and *Sufficient Working Capita*l play a role in deciding whether the financial environment favors a leveraged lease.

Structure

A Structured Matcher consists of a directed, acyclic graph of simple matchers. A simple matcher considers either the values of sub-decisions made by other simple matchers or the values of input data. Each data parameter feeds into only one simple matcher.

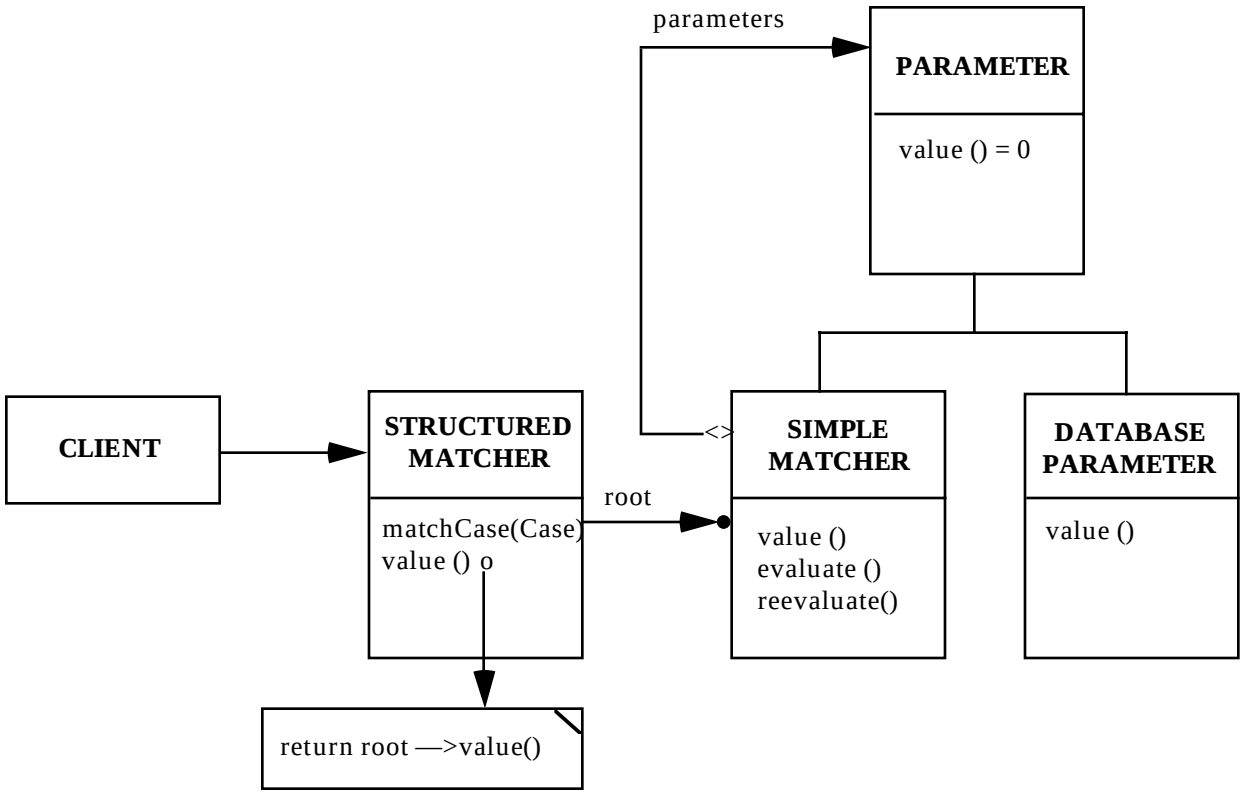

Figure 1. A Class Diagram for the Structured Matcher Pattern

A typical Structured Matcher looks like the hierarchy in Figure 2.

» A Structured Matcher for the leveraged lease decision might look like the hierarchy in Figure 3.

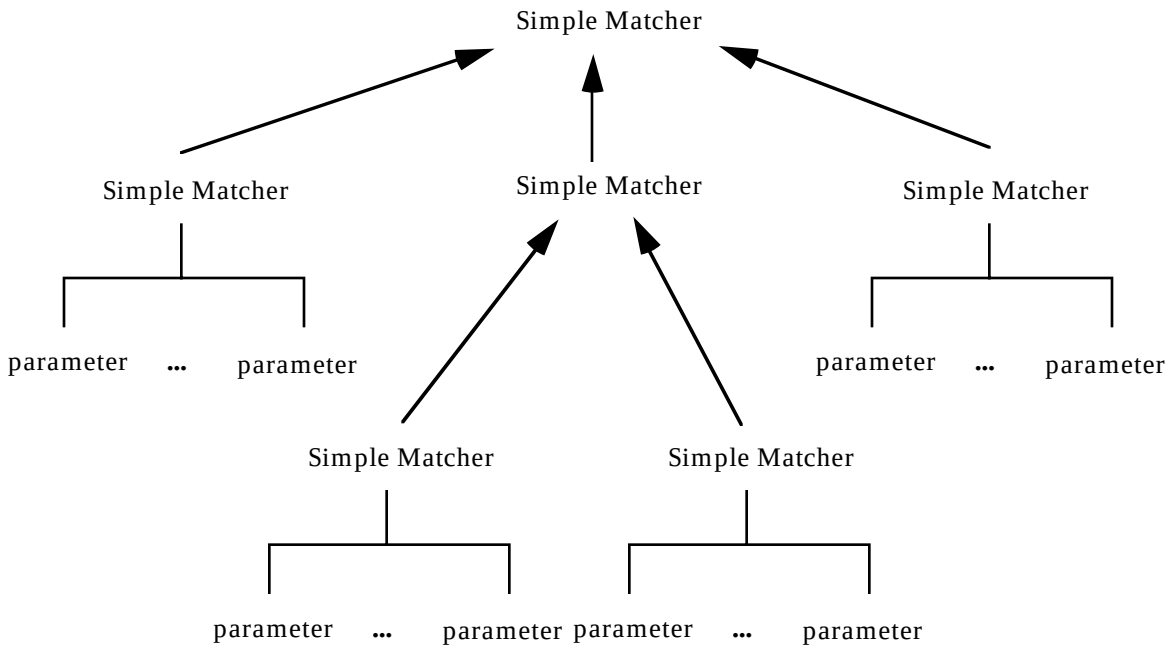

Figure 2. The Structure of a Structured Matcher

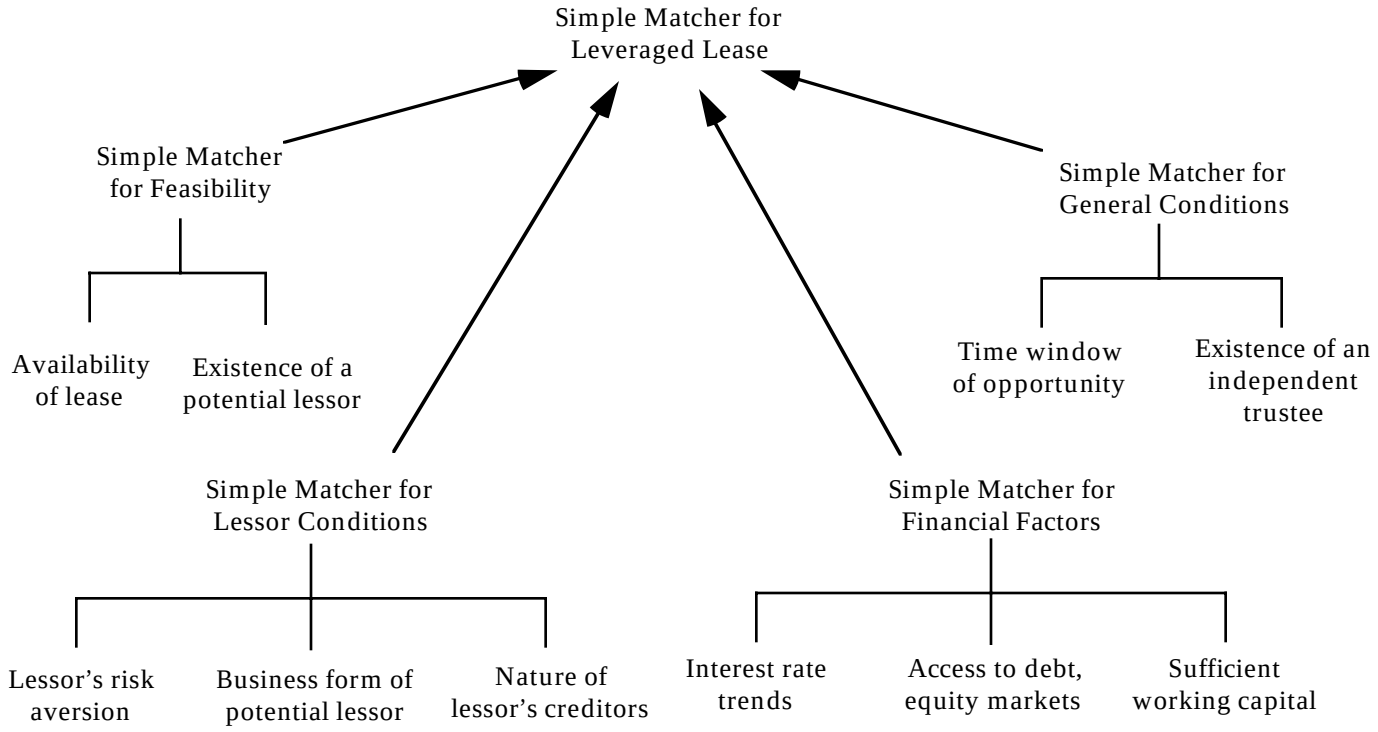

Figure 3. A Structured Matcher for the Leveraged Lease Decision

Dynamics

In a Structured Matcher, problem-solving control flows top-down from its root matcher to its leaves, and data flows bottom-up from the leaves to the root. (See Figure 4.) The client requests a value for the decision from the Structured Matcher, which delegates the task to its root simple matcher. The root makes its decision by requesting values for its sub-decisions whenever needed. Matchers at the leaves of the hierarchy request values for input data from the source of the data.

If a simple matcher can make its decision without considering all of its parameters, whether input data or sub-decisions, it will.

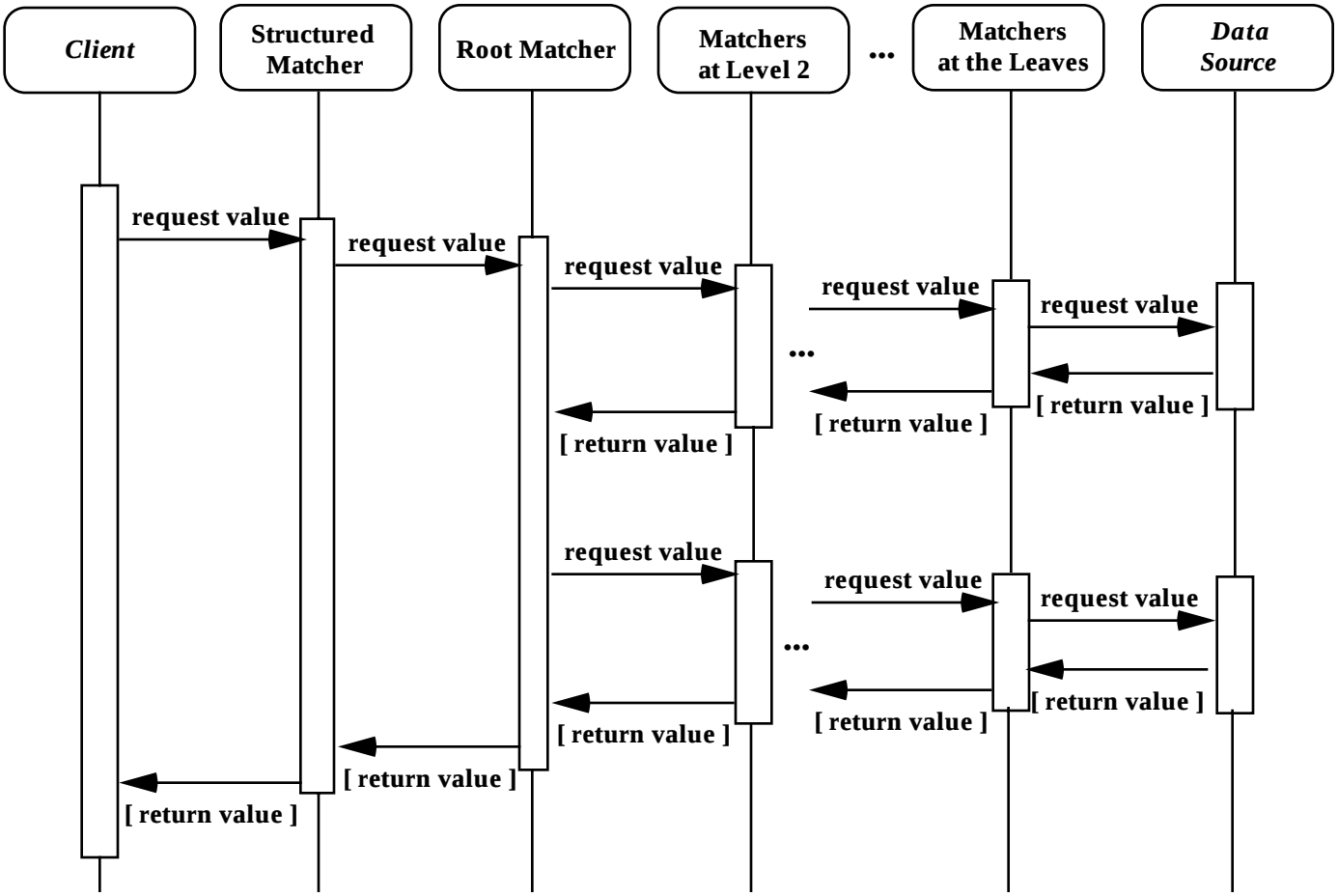

Figure 4. The Dynamic Structure of a Structured Matcher

» In the capital asset acquisition scenario, suppose that the user would like to know whether a leveraged lease is a good idea. The system would send a request to the leveraged lease Structured Matcher, which sends a request to the root matcher in its hierarchy. In order to make this decision, the root simple matcher

will ask some or all of its children for their values. It would then match these values against its rules to select its own answer, which would be returned to the requesting client.

The reason that the root matcher might not request values from all of its children is that, in some situations, a decision can be made on the basis of only one or two of the input parameters. For example, if a leveraged lease is not feasible, the Structured Matcher can return its answer without making its other sub-decisions. This kind of evaluation is called *short-circuit* evaluation.

Implementation

To implement a Structured Matcher, perform three primary tasks:

1. Identify meaningful sub-decisions.

   A sub-decision is part of the vocabulary of the domain, an abstraction of value to practitioners in the domain. Sub-decisions should make themselves apparent during the domain analysis phase of the system. Through recursive decomposition of decisions into sub-decisions, the structured matcher's hierarchy of simple of matchers will evolve.

   One way to approach this task is to partition the input data parameters to the decision along meaningful dimensions. These dimensions differ from domain to domain, but common criteria include causal relationship, close interdependency, and the source of data.

   As a general rule, each sub-decision should depend on between two and five parameters. As the number of input parameters increases, the worst-case complexity of the matcher grows exponentially.

   Likewise, the number of possible values—the size of the set from which the matcher chooses its answer—should be kept small, ordinarily no more than seven. As the number of possible values increases, the complexity of the matcher's *parent* grows. A larger set size is more acceptable when higher-level decisions can be made on the basis of a range of values, rather than on the basis of individual values.

   » In the leveraged lease example, decision decomposition and parameter grouping are based on how human experts in the domain reason. They first determine if the arrangement is feasible by examining the availability of leases and potential lessors. If the arrangement is deemed not feasible, a decision is returned. They then determine if one or more of the potential lessors is qualified, based on specific features of the lessors. If no qualified lessor exists, a decision is returned. Finally, they consider environmental factors and financial factors that make a leveraged lease more or less attrac-

tive. The answer they return depends on patterns of values for these two sub-decisions.

2. Implement each simple matcher.

   Decisions on Task 1 limit the number of inputs and the number of possible output values for the matcher, which limits the potential complexity of the decision. As a result, the programmer has some freedom when it comes to implementing each matcher. Among the options available are:

   - A decision tree.
   - A linear combination rule.
   - A set of pattern-matching rules.

   A common implementation uses the Table Matcher pattern, which is not documented in detail here. A Table Matcher consists of an ordered set of pattern-matching rules. To build a table matcher, identify a set of rules that map input data patterns onto the matcher's value set. Group rules according to their output value, and try to identify generalizations of the input patterns within each group. Use range tests on parameter values and don't-care matches in order to combine rules with common patterns into single rules. Once the set of rules is minimal and general, order the rules in a way that seeks to balance two forces: (1) the matcher should compute its answer efficiently, and (2) the matcher should consider its parameters in an order that makes sense within the domain.

   » In the leveraged lease example, the table for the root matcher might look something like Figure 5 (top of next page).

   The use of don't-care matches will make it possible for a matcher not to consider all of its parameters in all cases, depending upon the input data that it receives. The last rule in the table, whose pattern consists only of don't-care matches, is a default rule. If all preceding rules fail, the last rule will match and returns a default value for the choice.

3. Determine from where simple matchers at the leaves of the acquire their input data.

   The parameters of matchers at the leaves are basic data, not the results of sub-decisions made by other simple matchers. Typically, the matcher will request values for data parameters from some database or some other software component external to the Structured Matcher. This component hides the matcher from the details of data acquisition, data abstraction, and so on. In simple cases, the simple matcher may communicate directly with an interactive user of the system.

PATTERN				CHOICE
Feasibility	Lessor	General	Financial	
not favorable	don't care	don't care	don't care	highly unfavorable
don't care	not favorable	don't care	don't care	highly unfavorable
don't care	don't care	somewhat favorable or better	somewhat favorable or better	highly favorable
don't care	don't care	not unfavorable	not unfavorable	somewhat favorable
don't care	don't care	not favorable	not favorable	highly unfavorable
don't care	don't care	don't care	don't care	uncertain

Figure 5. A Table for the Root Matcher in Leveraged Lease

» In the capital asset acquisition system, the leveraged lease Structured Matcher sends all requests for primitive data values to a common database shared by all of the Structured Matchers and other components of the system. This both insulates the matchers from interface issues and allows the system to acquire each piece of data only once.

You will also want to consider these secondary issues:

1. Is the hierarchy of simple matchers a strict hierarchy (a tree) or a tangled hierarchy (a directed acyclic graph)?

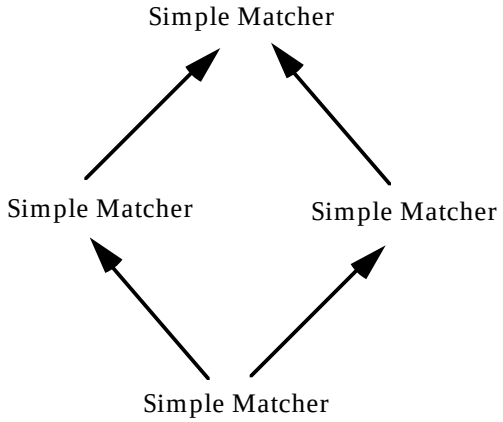

Figure 6. A Simple Tangled Hierarchy

In a tangled hierarchy, the decision of one simple matcher may serve as an input parameter to more than one higher-level matcher.

The advantages of a tangled hierarchy lie in its reuse of existing components within the Structured Matcher. The disadvantage of a tangled hierarchy lies in its potential computational cost. If the hierarchy is tangled, then matcher faces both exponential time complexity and exponential size complexity in the worst case. However, the worst case occurs only when the Structured Matcher contains an exponential number of simple matchers. In the average case, the number of simple matchers is linear in the number of parameters. As a result, a Structured Matcher can allow a considerable amount of tangling before encountering unacceptable performance. (Bylander et al., 1991)

2. Do simple matchers cache their answers?

   If a simple matcher caches its value, then it can respond more efficiently to future requests for its value in the same data context. This can be especially useful if the Structured Matcher consists of a tangled hierarchy. A simple matcher that caches its value must provide a second way to request its value, so that the Structured Matcher can force it to re-compute in a new data context.

Consequences

Structured Matcher offers the following benefits:

- *Computationally tractable*. By decomposing its task, a Structured Matcher manages the potential complexity of finding a solution. Limitations on the number of input parameters and number of possible values mean that exponential complexity can be constrained within small components.

- *Explanation at different levels of abstraction*. By decomposing its decision into sub-decisions, and perhaps decomposing those sub-decisions further, a Structured Matcher provides a vocabulary of intermediate abstractions in which to explain the system's answer.

- *Recognition of interactions among parameters.* By grouping parameters that are related to one another, the Structured Matcher pattern localizes interactions among parameters. Explicit representation of such interaction helps to reduce computational complexity and supports modification of the system.

- *Ease of modification*. Decision making is localized within simple matchers. When pattern-matching rules need to be changed or added, it is easier to identify the relevant part of the system. When input data para-

meters need to be modified, it is easy to identify which part of the decision they affect.

- *Facilitation of domain analysis and knowledge acquisition*. This pattern provides not only a way of thinking about system development, it also provides a way of thinking about domain analysis. As decisions are recursively decomposed, they introduce a vocabulary for talking about the problem, for organizing data parameters, for identifying gaps in domain understanding, and the like.

Structured Matcher may also impose some liabilities:

- *Potential brittleness in boundary conditions*. Many of the advantages of a Structured Matcher are predicated on the requirement that simple matchers consider a small number of input parameters and select from a small number of possible values. Unfortunately, this feature may cause the matcher to encounter *boundary problems*. As noted earlier, if a value for one of a matcher's parameters is unavailable at run-time, the matcher may be unable to make the correct decision, even if all the rest of its parameters match perfectly. More generally, this problem arises in the face of data is that merely erroneous, say, due to noise in a data stream.

  Structured Matchers are especially prone to erroneous data problems because they operate at a coarse level of "granularity". If a data parameter can take only two values (say, 'yes' and 'no'), then a potentially large number of data patterns may lie near the two boundaries between different values. By increasing the number of possible values of the parameter (say, to 'yes', 'maybe yes', 'uncertain', 'maybe no', and 'no'), the granularity of the system becomes smaller, decreasing the chance that an erroneous value will carry a wildly incorrect meaning. In the interest of computational tractability, Structured Matcher encourages relatively large granularity, thus exposing itself to boundary problems in some situations. In such cases, relaxing the ordinary restriction on the size of the set of possible values may be necessary. Done sparingly, this should not have an inordinate impact on the computational complexity of the matcher.

Example Resolved

Using a Structured Matcher resolves the forces at play in the leveraged lease decision. In particular:

- Ten parameters play a role in this decision, and each parameter can take five different values. In the worst case, a simple matcher requires $5^{10} \approx$ 10,000,000 pattern-matching rules. But the Structured Matcher of Figure 2 requires at most 925 rules.

- The Structured Matcher can explain its answer in terms of feasibility, conditions on the lessor, financial factors, and general conditions. Decisions for each of these can in turn be explained in terms of the relevant data parameters.

- By grouping related parameters, this Structured Matcher pattern can capitalize on interactions among them. The table in Figure 4 uses patterns about feasibility and lessor conditions to prune unfavorable situations and then focuses on interactions among general and financial conditions.

- Decomposing the decision into a hierarchy of sub-decisions makes the Structured Matcher more amenable to modification, both for debugging and for long-term maintenance. For instance, if the laws governing the timing and oversight of leveraged leases change, only the patterns in the simple matcher for general conditions need to be modified.

- The same intermediate decisions that allow the Structured Matcher to explain its answer also assist in domain analysis and knowledge acquisition, where they provide a vocabulary for talking about the leveraged lease decision.

Known Uses

Samuel (1967) described what is believed to be the first documented use of Structured Matchers in his landmark paper on machine learning in a checkers-playing program. His *signature tables* played an important role in making the learning component of the program computationally tractable.

In the early 1980s, Chandrasekaran and his research group began to study Mycin, the prototypical diagnostic expert system. The result of their analysis was a set of expert system patterns called *generic tasks* and MDX, a diagnostic expert system based on these patterns. They then generalized MDX into CSRL, a programming language for building hierarchical classification systems. CSRL used Structured Matchers to make decisions about diagnostic hypotheses. Chandrasekaran (1986) coined the term “hypothesis matching” and documented it as an independent technique. Finally, the group created Hyper, a programming language specifically for writing Structured Matchers.

Commercial use of Structured Matchers in generic task-based classification systems of this era included WELDEX, a system that detected welding defects (Mahalingham and Sharma 1985), and ROMAD, a system that did machinery vibration analysis (Mahalingham, Sharma, et al. 1985).

Ribar (1990) described LoanProbe, a program to evaluate the collectibility of commercial loans, used as a part of the internal audit practice at Peat Marwick, one of the Big Six accounting firms in the United States. LoanProbe is the most

widely documented large application of structured matching, consisting of approximately 9000 rules in thirty-three separate "knowledge bases" and interacting with sixty external programs and two data bases.

Mockler and Dologite (1990) used a cognitive modeling technique like Structured Matcher to teach an MBA course on strategic planning and management decision making as part of a larger research project. One result of this project was that their students, business professionals working full-time in industry, developed over 160 prototype systems using Structured Matchers. Follow-up surveys indicated that 28% of these systems were deployed and used by the developers' firms.

Mitri (1991) used candidate selection to build several systems through the Center for International Business Education and Research at Michigan State University. MAPS (Mitri 1995) is a more recent application, in the domain of market entry planning.

Wallingford and Sticklen (1992) documented Acquisition, a system for planning capital asset acquisitions, from which the running example in this paper is drawn. In Acquisition, Structured Matchers are used to evaluate the fitness of each transaction type in a hierarchical classification scheme.

More recent work in the generic task community used Structured Matchers in more diverse roles. Wallingford (1998) documented several known uses of the Sponsor-Selector pattern, which typically uses Structured Matchers in two roles: resource sponsorship and resource selection. One example is Router (Goel et al., 1994), a robot navigation path planner that uses Structured Matchers to select among planning methods at run-time.

Variants

QBKG. The QBKG backgammon program (Berliner and Ackley, 1982) used a hierarchical position evaluation function reminiscent of the Structured Matcher pattern. QBKG estimates the strength of a potential move by combining the values computed for more specific features. These values are computed from more primitive features whose values can be assessed directly. This approach resembles Structured Matcher in its hierarchical abstraction of features but differs in that it never maps feature values onto a set of discrete values. Instead, QBKG uses a variety of arithmetic combining functions on continuous values. This approach trades the advantages of Structured Matcher's discrete representation, including enhanced explanation capability and computational simplicity, for more robust performance in the face of incomplete or erroneous input data. Another drawback of this approach is that linear combination functions cannot account for direct interactions among feature values as flexibly as discrete representations can.

Candidate selection. Candidate Selection (Mitri 1991) also generalizes Structured Matcher to work with continuous values and combination mechanisms such as weighted sums. Mitri recognized that the robustness of QBKG's approach in the face of boundary conditions resulted not from its continuous representation but from its use of a compensatory scoring mechanism that dampened the effect of individual data items. By incorporating a compensatory mechanism, Candidate Selection hopes to provide the advantages of both the Structured Matcher pattern and the QBKG approach.

See Also

The **Sponsor-Selector** pattern (Wallingford 1998) typically uses Structured Matchers for both recognition and selection tasks. Each sponsor uses a Structured Matcher to determine if its resource applies in the current context. The selector uses a Structured Matcher to select from among the resources that do apply.

The hierarchy of simple matchers in a Structured Matcher follows the **Composite** pattern (Gamma et al. 1995). The intent of Composite is to allow client code to treat individual objects and compositions of objects in the same way. Here, the client of the Composite is not the system client but the Structured Matcher object, which provides the public decision-making interface. The Composite is hidden from the system client. Further, the Structured Matcher pattern focuses on the behavior of the system, rather than its structure. The key to a Structured Matcher lies in how the responsibility for making a decision is distributed across simple matchers.

Influence diagrams have a long history of use in accounting and auditing. The use of Structured Matchers by Ribar (1990) and Mockler (1990) grow out of this tradition, but both have extended the idea in ways that approach the Structured Matcher pattern.

One of the important benefits offered by Structured Matcher lies in its use for domain analysis. Ribar (1990) describes how the pattern can be used in analysis. First, the analyst elicits aspects of a situation that affect a decision. Next, the analyst culls the set of aspects, identifies overlaps, and organizes them in a hierarchy of sub-decisions. Finally, the analyst works with the domain specialists to build a decision table or other simple matcher for evaluating each aspect. In artificial intelligence, this task of *knowledge acquisition* is a primary bottleneck in the software life cycle, and support for the process is actively sought. Ribar built his technique into the Knowledge Acquisition Tool, a commercial system that allows non-programmers to build systems that use the Structured Matcher pattern.

Finally, Bylander et al. (1991) wrote the primary research article on the structured matching technique and did foundational theoretical analysis of its properties. They formally define the task of a Structured Matcher and analyze the computa-

tional complexity of its solution. Their paper documents several other known uses of the pattern and several other techniques related to structured matching.

Acknowledgements

My advisors at Michigan State University, Jon Sticklen and Bill Punch, introduced me to structured matching and helped shape my views of this pattern. I thank Mike Mitri for discussing Structured Matching and Candidate Selection with me many times during our time together at MSU. I also thank Gerard Meszaros, my PLoP'98 shepherd, for his help in improving the paper.

References

1. Berliner, Hans J., and D. H. Ackley. The QBKG System: Generating Explanations from a Non-Discrete Knowledge Representation. In *Proceedings of the National Conference on Artificial Intelligence*. Pittsburgh, pages 213-216, 1982.

2. Brown, David C., and B. Chandrasekaran. *Design Problem Solving*. Morgan Kaufman, San Mateo, California, 1989.

3. Bylander, Tom, Todd R. Johnson, and Ashok Goel. Structured Matching: A Task-Specific Technique for Making Decisions. *Knowledge Acquisition* 3:1-20, 1991.

4. Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture*. John Wiley and Sons, New York, 1996.
5. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, New York, 1995.

6. Goel, Ashok, Khaled Ali, Michael Donnellan, Andres Gomez, and Todd Callantine. Multistrategy Adaptive Navigational Path Planning. *IEEE Expert*, 9(6):57-65, December 1994.

7. Mahalingham, S., and D. D. Sharma. WELDEX—An Expert System for Nondestructive Testing of Welds. In *Proceedings of the Second Conference on Artificial Intelligence Applications*. IEEE Computer Society Press, New York, pages 572-576, 1985.

8. Mahalingham, S., D. D. Sharma, B. Eliis, and B. Morrow. ROMAD: An Expert System for Vibration Analysis. In *Proceedings of the Control West Conference*. ?????, Long Beach, 1985.

9. Mitri, Michel. A Task-Specific Problem-Solving Architecture for Candidate Evaluation. *AI Magazine*. 12(3):95-109, Fall 1991.

10. Mitri, Michel. MAPS: A Planning System for Market Entry Planning. In *International Journal of Intelligent Systems for Accounting, Finance, and Management*. 5:2, June 1996.

11. Mockler, Robert J., and Dorothy G. Dologite. Knowledge-Based Systems Applications for Business Decision Making. Working Paper, Business Research Institute, St. John's University, May 1990.

12. Ribar, Gary S. The Representation of Cognitive Task Performance. In *Proceedings of the Third International Symposium on Expert Systems in Business, Finance, and Accounting*. Graduate School of Business Administration, University of Southern California, 1990.

13. Samuel, Arthur. Some Studies in Machine Learning using the Game of Checkers. *IBM Journal* 11:601-617, 1967.

14. Wallingford, Eugene. Sposnor-Selector. In *Pattern Languages of Program Design 3.* Edited by Robert Martin, Dirk Riehle, and Frank Buschmann. Addison Wesley, Reading, Massachussetts, pages 67-78, 1998.

15. Wallingford, Eugene, and Jon Sticklen. Selecting Plans for Capital Asset Acquisition through Classification Problem Solving. In *Expert Systems in Finance*. Edited by Daniel E. O'Leary and Paul R. Watkins. Elsevier Science Publishers, Amsterdam, pages 19-36, 1992.