Automated Software Warehouse Management

STUART H. RUBIN
SPAWAR Systems Center
San Diego, CA 92152-5001, USA

WEI DAI
School of Information Systems
Victoria University
PO Box 14428, Melbourne, MC 8001
Victoria, Australia

Abstract: - This paper proposes a knowledge-based approach to manage software warehouses. It is understood that knowledge acquisition is the bottleneck for intelligent systems of all kinds. Our research focuses on solutions for both theoretical and practical aspects of the bottleneck tasks through the proposed mechanisms of randomization, symbolic representation, and grammatical inference.

Key-Words: - Knowledge Acquisition, Symbolic Representation, Randomization, Software Warehouse, Grammatical Inference, Transformation.

1 Introduction

Based on the concepts and roles of traditional data warehouse, the concepts of software warehouse were raised [1]. Software warehouse is a special store where hundred or thousands of software modules are stored, accessed, and potentially packaged into meaningful software applications. This paper investigates a knowledge based approach for the automated management of software warehouses. The knowledge-based approach is intended to produce packaged solutions i.e. software marts that are based on the concepts of traditional data marts. A software mart is a repository of software assets gathered from operational software sources to serve a specific application purpose. The key for successfully managing the assets comes from the knowledge that directs the transformation and grammatical inference to manipulate the concerned software assets and their packaging processes.

2 The Process of Packaging

Suppose we have previous developed software assets in the already classified software libraries or software warehouses in the modules of G1, G2, G3. Each module consists of a list of potentially reusable software parts (e.g. implemented objects). The software assets are described as the following:

G1 -> g11, g12, g13
G2 -> g21, g22, g23
G3 -> g31, g32, g33

Each software asset is further described by the following metrics. For instance,

g11 -> g11S, g11P, g11R, g11I (considered as list of the properties for the asset and the following symbols were used: S – Scope, P – Purpose, R – Role, I – Interface).

Each property is further decomposed into attribute-value pairs. For instance,

g11S -> scope, data management
g11P -> purpose, extraction
g11R -> role, a primitive function
g11I -> interface, (in: x, y, z; out: g, w)

The software assets are designed to be orthogonal. Each of the specific software assets, i.e. functions,

such as g11, g21, g311 etc., are associated with an arbitrary number of features (e.g. attribute value pairs) that best describe the concerned functions. The creators of the software can specify the positive and negative examples of the asset's metric (i.e. scope, purpose, role etc) to assist in eventual asset retrieval at a later stage. The related software assets form the software patterns. These patterns are further classified as internally and externally stable intangible objects as were used in [2], [3]. The objects deal with the conceptual aspects (which are reusable) of software development tasks and are identified by their respective roles. The roles describe the interface aspects of the objects. The intangible layer of the reusable object supports the adaptation and substantiation of tangible objects which are internally adaptable and externally stable following modern object-oriented software engineering principles. The tangible objects are packaged through the use of glue codes. The glue codes describe the behavioural aspects of the objects. The roles and glue codes together form the connectors of software components. A general description for the packaging process (the key for application construction) is shown in Figure 1.

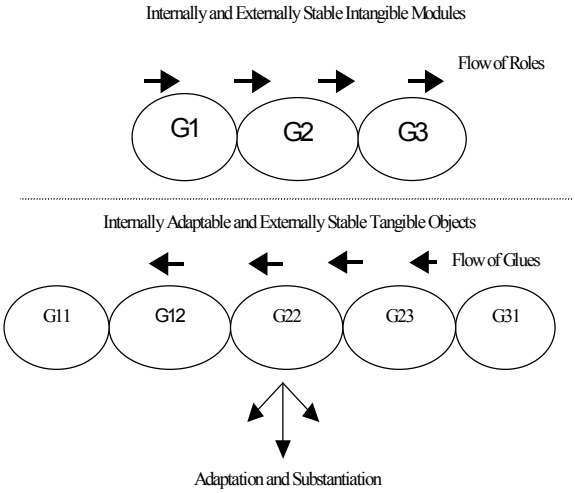

Figure 1: Reusable Assets Identification and Packaging

In order to support the degree of automation on the software warehouse, knowledge needs to be acquired. In the following sections, we discuss the bottleneck knowledge acquisition issues, as well as the processing and manipulations of software assets through symbolic representation and grammatical inference activities.

3 The Knowledge Acquisition Bottleneck

The knowledge acquisition bottleneck refers to the difficulty of capturing knowledge for use in the system. Whether the system is used for evaluating bank loans, intelligent tutoring, or prescribing medical treatments, the question remains: How do we obtain the knowledge used by the system (i.e., code it) and how do we verify it for mission critical applications. For example, the medical rule, IF the patient is coughing THEN prescribe cough syrup; is usually true. However, an expert medical system having this limited medical knowledge would prescribe cough syrup to someone whose airway is obstructed during the course of a meal say! It is lacking in the more specific rule, IF the patient is coughing AND the patient was just eating THEN apply Dr. Heimlich's maneuver with certainty factor = 0.90.

We see from this one small example that knowledge acquisition needs to be relatively complete, lest there be potentially dire consequences. Now suppose that you are a knowledge engineer whose task is to write rulebases -- each containing thousands of rules for say mission critical applications (e.g., flight-control systems, medical decision support systems, weapons systems, vehicle guidance systems, and many more). It should be clear that the task is too difficult for human beings given the current state of the art in knowledge acquisition. It is just that this software is inherently difficult to test.

We will propose a radically new method for cracking the knowledge acquisition bottleneck; but, before we do, let us see what the problem really is. First, it should be mentioned that we have evolved higher-level languages because they are more closely aligned with the way in which we think. This should come as no surprise. Indeed, the Windows or other GUI was developed as a metaphor for our spatial-temporal reasoning. After all, we cannot be easily taught to read machine code for we are not machines.

3.1 Mining for Rules

Data mining also fits into this unified theory of intelligence. Solomonoff has done much work in the area of grammatical inference [4], [5]. This is the induction of grammars from examples. It has been proven that for context-free grammars or higher, induction must be guided by heuristics. In essence, grammatical inference is the randomization of data. Given that we succeed in formalizing an expert

system having automated knowledge acquisition facilities in the form of context-free grammars, it follows that data can be mined to create such expert systems by building on the work of Solomonoff. Note that if the mining process is itself directed by expert systems, then it is called directed mining. Just as conventional expert systems, such as XpertRule (i.e., which we rate at the top of the line), have data mining tools available, so too will our strategic expert systems. They will be implementations of Kurt Gödel's and Gregory Chaitin's theories of randomness [6] and Roy Solomonoff's related work on grammatical inference [4], [5].

4 Randomization as a Measure of Intelligence

An intelligent system interacts with the user in two manners. First, it requests random knowledge be supplied where necessary. Second, it asks the user or knowledge engineer to confirm symmetric knowledge where presented. Note that supplying a selection from a pull-down menu is partially random and partially symmetric in its component tasks.

Clearly, if a knowledge engineer can supply the requested random or symmetric knowledge, then it is through the application of acquired knowledge. It follows that if that knowledge can be captured in a knowledge-based system(s), then the requested tasks can be automated. Furthermore, let our grammar-based system be used as the shell, which learns the knowledge that would otherwise be requested of the user or knowledge engineer. An interesting and inescapable conclusion follows. That is, the only thing not automated would be the acquisition of random knowledge by the auxiliary system(s). In other words, randomness can be defined along a continuum in degrees. What is being claimed is that a network of cooperating grammar-based systems requests knowledge that is random in proportion to the size of its collective randomized knowledge bases.

As more and more knowledge bases are linked in a network, then the knowledge needed becomes more and more random. For example, the following indefinite sequences are observed to occur in increasing order of randomness: 1, 1, 1, 1, 1, 1, 1, 1, 1, 1; 1, 1, 2, 1, 2, 3, 1, 2, 3, 4; 1, 4, 2, 7, 5, 3, 6, 9, 0, 8. That is, the generating function for each successive sequence grows larger and larger until the sequence is a fixed point, or truly random. Consider the limit. Here, the knowledge bases reach some critical finite radius where the knowledge embodied cannot be bounded (e.g., as in a finite number of rules). At this point, the needed knowledge is truly random. Random knowledge is analogous to the Schwarzchild radius of a black hole, beyond which nothing -- not even radiation -- can escape the gravitational pull [7]. For example, the system may benefit from knowing that coherent light is polarized when reflected off of a magnetized surface, or that when heated to the Fermi point iron loses all magnetic properties, etc. This exemplifies random knowledge. It follows that an example of symmetric knowledge is that coherent light is oppositely polarized when reflected off of a magnetized surface having the opposite N-S polarity.

A consequence of Gödel's Incompleteness Theorem [6] is that countably infinite truths are recusively enumerable, but not recursive. Thus, the universe will forever hold the most widely varied knowledge to be discovered. Indeed, as the early 20^{th} century mathematicians came to discover, this is a reason for rejoicing for it is what makes life interesting!

Consider two total computable functions, f, and g. We say that they are total because they are defined over all members of the domain. They are said to be computable because they can be realized by some algorithm. In particular, let that algorithm be substituted for by a sequence of transformation rules, which may be iterated over. Moreover, let A and A' represent a pair of symmetric domains. For example, A could represent an increasing sort and A' could represent a decreasing sort. Let B represent an orthonormal or mutually random domain with respect to A. For example, B could represent an algorithm to invert a matrix. Then, $f(A) \rightarrow A'$ and $g(A) \rightarrow B$. It follows that $|f| < |A'|$ and $|g| \geq |B|$. These relations follow because the fixed point for the magnitude of the transformational map is the magnitude of the image of transformation itself. That is, one can always replace the rule base, g, with B' such that $B'(A) \rightarrow B$ where $|B'| \sim |B|$. This means that mutually symmetric domains are characterized by rulebases consisting of fewer, shorter, and more reusable rules than are the more random pairs. The larger are the domains in the pairing, the more likely are they to embed symmetric constructs. We already know that the world is neither totally random, nor totally symmetric. Indeed, this follows from Gödel's Incompleteness Theorem. The degree of symmetry increases with scale. Were this not the case, then the universe would be random in the limit. (If the degree of symmetry didn't change with scale, then the notion of chaos would be violated.) Thus,

$\left(\lim|A| \to \infty \frac{|A'|}{|f|} = \lim|A| \to \infty \frac{|B|}{|g|} \right) \to \infty$. In other words, the degree of randomization possible is in proportion to the magnitude of the information, where there is no upper bound. Also, the processor time required to randomize such information is unbounded. Such absolute minimal entropy can of course never be achieved for it would violate the Incompleteness Theorem. However, there is nothing to preclude the construction of randomizing systems to any desired level of utility.

5 Symbolic Representation

This section will bring together the previous ones as we begin to develop an algorithm for realizing an intelligent system. Let, Qi be domain-specific questions that are collectively organized into hierarchies. Think of nested pull-down menus. The first menu details the top of the hierarchy (e.g., chemistry). The next submenu under chemistry say offers the choice of inorganic chemistry. Under the inorganic chemistry menu we have salts and under salts we have domain-specific questions, which determine the antecedent predicates. For example, a sample question here would be, "Is the salt a chloride?" or "Enter the chemical formula for the salt:". Replies are posted to the blackboard for subsequent processing. Conflict resolution will favor the most specific rule.

The consequent predicates are organized just as are the antecedent predicates with one notable difference: Instead of questions, which define the antecedent predicate primitives; actions, functions, or methods define the consequent primitives. For example, a sample action here would be, "Dissolve the salt in triple-distilled water," or "dissolve ()". Predicate definitions are saved in symbol tables external to the grammar in the interests of efficiency.

Let us adopt the standard that different levels in the hierarchy will be separated by dots (.). That is, a dot separates a subclass from its superclass. An arbitrary number of levels is allowed. Successive integers, starting with one, will be used for the representation of all predicates. The same integers may be reused at different levels. A simple lookup table provides for their translation to text or effective code as designated. A distinct grammar is used for the antecedent predicates and the consequent predicates. The zero integer is reserved, since it has no negation. Here are sample primitive lookup tables:

Antecedent	Grammar	Consequent	Grammar
1	Chloride?	1	Add water.
2	Formula:	2	Dissolve ()

Table 1. Primitive Symbol Tables

The symbol 1 represents the question, "Is the salt a chloride?" in the antecedent grammar and the action, "Dissolve the salt in triple-distilled water," in the consequent grammar. The use of the symbol 2 is similar.

The notation, i.j represents the ith superclass, which contains the jth subclass. We will adopt the convention that j.k aligns with j.k in i.j.k (i.e., right-justified). This may be termed, *subclass justification*. The rightmost subclass is defined to be a primitive. Primitives cannot be subclassed by definition. This leads to an expanded definition for our symbol table, as follows.

Antecedent	Grammar	Consequent	Grammar
1	Chloride?	1	Add water.
2	Formula:	2	Dissolve ()
1.*	Salts	1.*	Salts
1.*.*	Inorganic	1.*.*	Chemistry

Table 2. Hierarchical Symbol Tables

Subclass justification implies that primitives are always uniquely identified by a single integer. Again, separate tables are maintained for the antecedent and consequent grammars because the hierarchical definitions may be distinct. The asterisk is used to denote any subclass.

We will use commas to separate multiple items at the same level. Negative numbers mean "not". For example, 1.-1,-2 in the antecedent grammar is interpreted to mean all salts, excluding the chloride and excluding the formula questions. The pull-down menu for Salts would then be empty. A lexicographic order (i.e., -1,-2 rather than -2,-1) is imposed to facilitate the pattern matcher. Similarly, 1,-2 is interpreted to mean including the chloride question, but excluding the formula question in the antecedent grammar. Again, this interpretation is used to determine the entries in the pull-down menus at each level of the hierarchy. That is, use positive integers if relatively few entries are to be included and use negative entries if relatively few entries are to be excluded. This itself is a form of

randomization. The interpretation of the consequent grammar is similar.

6 Grammatical Inference

The next step is to tie the above together by way of grammatical inference. To begin with, let us write a pair of domain-specific rules and see how they can be randomized:

R1: IF the metal is Sodium ∧ the gas is Chlorine THEN the salt is NaCl
R2: IF the metal is Potassium ∧ the gas is Fluorine THEN the salt is KFl

Now, these are the system rules, but we need to transform these rules into a set of questions to elicit the required knowledge from the user or knowledge engineer in a directed manner. The actions are reduced to integers.

Q1: What is the metal having a +1 valence?
Q2: What is the reducing gas having a -1 valence?
A1: NaCl
A2: KFl

The rules are now:

R1: IF Q1 = Sodium ∧ Q2 = Chlorine THEN A1
R2: IF Q1 = Potassium ∧ Q2 = Fluorine THEN A2

Here are the associated symbol tables:

Antecedent	Grammar	Consequent	Grammar
1	Sodium	1	A1
2	Potassium	2	A2
3	Chlorine	3	Table salt
4	Fluorine	4	Potassium Fluoride
1.-3,-4	Q1	5	Salt
2.-1,-2	Q2	1.*	Salts

Table 3. The Chemical Symbol Tables

Notice that the (antecedent) integer hierarchies also serve as a data dictionary (e.g., 1.3 or 1.4 would be disallowed). Now, we are in a position to define a rudimentary grammar. We deviate from traditional grammar notation in that the reductions are to be taken to the right. This is done to be synergistic with the appearance of the rules with the rule antecedent on the left and the rule consequent on the right. The antecedent grammar follows.

1.1 & 2.3 → 1.1
1.2 & 2.4 → 1.2

The consequent grammar follows.

1.1 → 1.3 | 1.5
1.2 → 1.4 | 1.5

Here is how the grammars would operate: If you were to start by asking Q1 and answering with the metal sodium, then the system would ask you if the reducing gas were chlorine. If so, then it would inform you that you have table salt, or salt. If not, then the system would superclass chlorine and ask you for another reducing gas having a -1 valence. Since the data dictionary tells us that this can only be chlorine or fluorine, then it in effect asks you if the reducing gas is fluorine. If not, then the system would direct you to specify and classify a proper gas, if any. If so, then NaFl is as yet unknown by the system. That is, 1.1 & 2.4 does not match any antecedent in the segmented rule base. The knowledge engineer is thus asked to specify a consequent, which is also acquired in the consequent symbol table, if necessary -- say, 1.6. The same process occurs if a more general rule is matched (i.e., one whose antecedent is covered by the context), which produces an incorrect action(s). The new antecedent rule is thus:

1.1 & 2.4 → 1.6 | 1.5

A special start symbol, A, is introduced to define those rule consequents that are already primitives:

A → 1.6 | 1.5

Thus, the antecedent grammar now appears as:

1.1 & 2.4 → A
1.1 & 2.3 → 1.1
1.2 & 2.4 → 1.2

The updated consequent grammar now appears as:

A → 1.6 | 1.5
1.1 → 1.3 | 1.5
1.2 → 1.4 | 1.5

Observe from the antecedent grammar that if the metal sodium were first specified, then there would now be ambiguity in what to suggest next -- chlorine or fluorine. Here, fluorine would be presented first as it was part of the most recently acquired rule. Of course, if the metal potassium were first specified, then fluorine would automatically be presented first, since there is no ambiguity at this point. Notice again that the system has learned to improve its behavior on the basis of experience.

Next, these grammars are to be subjected to randomization operations -- each one distinct from the other. As it turns out, these grammars are already random. Thus, we introduce an abstract antecedent grammar instance and proceed from there. The consequent grammar is randomized similarly. Let the abstract antecedent grammar instance be defined as follows.

$$1.1 \ \& \ 2.2 \ \& \ 2.3 \rightarrow A$$
$$1.1 \ \& \ 2.2 \ \& \ 2.4 \rightarrow 1.1$$
$$1.2 \ \& \ 2.2 \ \& \ 2.4 \rightarrow 1.2$$

This instance can be randomized as follows.

$$B \rightarrow 1.1 \ \& \ 2.2$$
$$B \ \& \ 2.3 \rightarrow A$$
$$B \ \& \ 2.4 \rightarrow 1.1$$
$$1.2 \ \& \ 2.2 \ \& \ 2.4 \rightarrow 1.2$$

Notice now that the randomization operation is also *ambiguous*. This implies the need for heuristics in randomizing a context-free grammar. The following is an alternative randomization.

$$C \rightarrow 2.2 \ \& \ 2.4$$
$$1.1 \ \& \ 2.2 \ \& \ 2.3 \rightarrow A$$
$$1.1 \ \& \ C \rightarrow 1.1$$
$$1.2 \ \& \ C \rightarrow 1.2$$

Different degrees of randomization will be obtained depending on the order of reductions. Here, the B and C substitutions yield equivalent degrees of randomization. Exhaustive search is needed to minimize the resultant grammar. Fortunately, the grammar need not be minimized because the extra 1 or 2 percent compression does not improve performance any more than that and it does cost orders of magnitude more computing time.

Observe how the randomized grammars allow subsequences to be matched that need not match the prefix (e.g., $C \rightarrow 2.2 \ \& \ 2.4$). Here, given 2.2, 2.4 would be suggested in the absence of a more specific match. 2.3 would not be suggested because 2.2 requires 1.1 as a prefix for this case.

In practice, array-based pointers would point to Cs definition. The letter definitions are recursively subject to randomizations where possible. The inference engine will match the productions and where necessary superclass them. It is not necessary to maintain separate grammars for this purpose as the 'primitive' grammar is the most informative.

7 Conclusion

We have proposed solutions to overcome the knowledge acquisition bottleneck to support an automated software warehouse application. Several technical issues regarding the effective use of the acquired knowledge on software assets through symbolic representations were discussed. The solutions focus on representation, randomization, grammatical inference, and transformational aspects of the technical challenges.

References:

[1] H. Dai, W. Dai, G. Li. Software Warehouse: Its Design, Management and Application, *International Journal of Software Engineering and Knowledge Engineering*, 2004 (to appear).

[2] M.E. Fayad, Accomplishing Software Stability, *Communications of the ACM*, Vo. 45, No. 1, January 2001, pp 95-98.

[3] M.E. Fayad, and A. Altman, Introduction to Software Stability, *Communications of the ACM*, Vo. 44, No. 9, September 2001, pp 95-98.

[4] R. Solomonoff, "A new method for discovering the grammars of phrase structure languages," *Proc. Int. Conf. Inform. Processing*, UNESCO Publ. House, Paris, France, pp. 285-290, 1959.

[5] R. Solomonoff, "A formal theory of inductive inference," *Inform. Contr.*, vol. 7, pp. 1-22 and 224-254, 1964.

[6] G.J. Chaitin, "Randomness and mathematical proof," *Scientific Amer.*, vol. 232, no. 5, pp. 47-52, 1975.

[7] C.T.J. Dodson and T. Poston, *Tensor Geometry*, 2d ed., New York, NY: Springer-Verlag, Inc., 1997.