A Report On

Semantic String Matching

For the fulfilment of the course

CS F266 : Study Project

Submitted to :

Prof. Sundar Balasubramaniam

Department of Computer Science & Information Systems

Submitted by :

Rajath S

2012B5A7589P

Abstract

The problem of encoding semantic information into matching strings is explored. The syntactic structure of strings is specified using a Context Free Grammar. Methods to specify semantic rules as relationships between syntactic constructs is examined. An efficient algorithm to semantically match strings is constructed using hashing. The algorithm is implemented to match XML documents by inferring semantic rules from XML Schema. Further, a software is designed to aid domain experts in interactively creating XML schema for domain specific content.

Certificate

This is to certify that the report entitled, “Semantic String Matching” and submitted by Rajath S ID No. 2012B5A7589P in partial fulfillment of the requirements of CS F266 Study Project embodies the work done by him under my supervision.

Supervisor
Sundar Balasubramaniam
Faculty,
BITS-Pilani Pilani Campus
Date:

Table of Contents

Introduction

The objective of the project is to encode semantic information into string matching process. This problem raises several interesting theoretical questions such as language design for semantics specification, efficient representation and application of semantic rules for matching, adaptability of semantic rules specification to domains, limitations and error bounds (in case of approximate and/or randomized semantic matching).
Some applications of semantic matching include duplicate item detection, automatic answer evaluation systems, extracting useful information from incomplete data (such as forms, surveys), information indexing and retrieval, etc.

In this report, a limited approach to semantics specification using XML schema is described. An efficient matching algorithm using hash functions is outlined.

Related Work

Several domain specific application software related to semantic matching are available.

Coccinelle [1] is a code refactoring tool that provides a semantic patching language to specify matches in C code. This is used in the linux kernel development to find similar similar copies of a code snippet and transform them into functions. This helps in software maintenance and automatic modularization.

I.D. Baxter et al. [2], also propose a software maintenance tool using the idea of hashing to find clone abstract syntax trees.

Matching Process

Syntactic structure of the strings and semantic equivalence rules must be provided for matching two or more strings conforming to both syntactic and semantic rules.

Two strings P and Q are syntactically equal if and only if, |P| = |Q| and P[i] = Q[i] for all i=1..|P|. Based on the requirements of a domain, two syntactically unequal strings may be semantically equivalent. However, if two strings are syntactically equal, they are semantically equivalent by the strict definition of syntactic equivalence. For instance, if we consider the expression grammar, strings “x*y” and “y*x” are not syntactically equal. If the domain under consideration is natural numbers, then the strings are semantically equivalent. In case of Matrix algebra, this is in general not true as multiplication is a non-commutative operation. User input is required to define the notion of semantic equivalence, as it is clearly dependent on the domain and cannot be inferred from the grammar.

Syntax specification

We adopt Context Free Grammar as a tool to specify the syntax. Syntax describes the structure of the strings. Semantic rules require this structural description as they are expressed as relationships between different syntax constructs.

The language generated by a Context Free Grammar (CFG) G (∑, V, P, S), denoted L(G), is a set of all strings (using alphabet ∑), that have a valid derivation (restricted by production rules P), from the start symbol S. The set of context free languages is a proper superset of the set of regular languages. Every regular expression can be expressed as a left-linear context free grammar. Note that, Context Free Grammars have significant practical applications, including programming languages, data representation, natural languages, etc.

The process of verifying whether a string is generated by a CFG, is known as parsing. A parser produces a unique parse tree for each string generated by the CFG (if the grammar is unambiguous). A parse tree is a graphical representation of the derivation of the string using production rules of the CFG. The worst case asymptotic complexity of Parsing is ϶Θ(x^3 * |G|) using dynamic programming. However, for a restrictive class of CFGs (sufficient for most practical purposes), a linear time parsing algorithm can be constructed
In practice, LALR parsing is the preferred parsing method as it consumes lesser memory as compared to LR(0) parsing and covers a larger set of grammars as compared to SLR parsing.

To verify if a string can be generated from the CFG, parsing techniques discussed above are used.

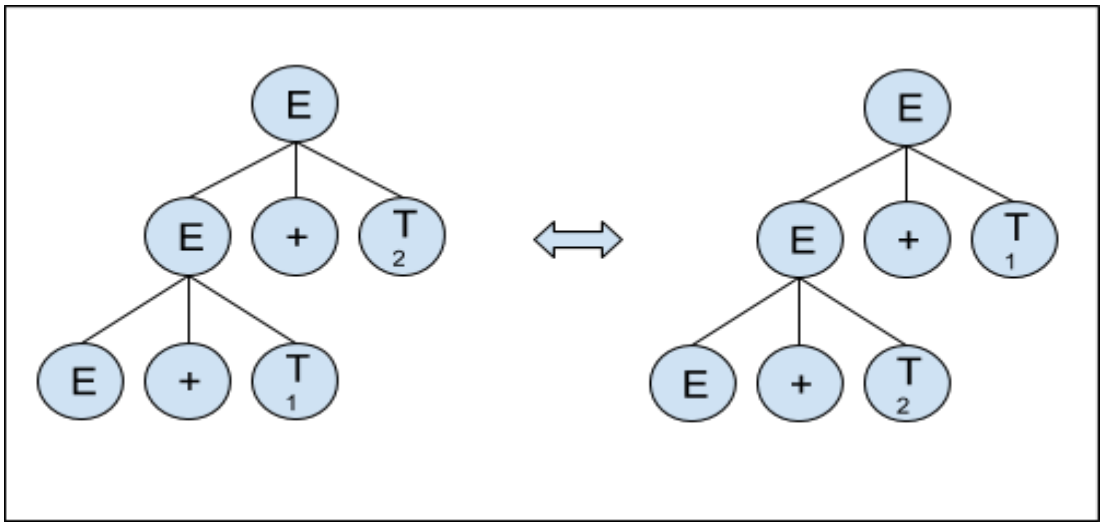
Semantic Rules

Semantic rules can be captured in terms of relationships between grammar symbols. To clarify, semantic rules can be expressed as pattern in the parse tree and a rewrite rule that can produce another semantically equivalent parse tree. Note that, application of the rewrite rule must not produce a parse tree that cannot be generated by the given context free grammar.

As an example, consider the expression grammar G (figure 1):

E → E + T | E – T | T
T → T * F | T / F | F
F → <id> | (E)

Commutativity of addition can be expressed as the following rewrite rule :



Note :
[1] Unit productions have been suppressed in the above diagram.
[2] Numbers label multiple occurrences of the same non-terminal.

A Parse tree is a naturally recursive structure. And so the rewrite rules can be repeatedly applied to obtain several semantically equivalent strings. Also note that, associativity of +, –, *, / , distributivity of * over +, etc., can be expressed with similar tree rewrite rules.

Tree rewrite rules are not the most general way to express semantic rules. There are certain semantics that cannot be captured using tree-rewrite rules. Tree-rewrite rules are local. Any global semantic rule cannot be expressed as a tree-rewrite rule. For example, semantic equivalence of isomorphisms of a graph (string isomorphism in our case) cannot be expressed as a tree-rewrite rules.

Another problem with tree rewrite rules is that the number of such rules required to express the semantics may become very large due to combinatorial explosion. This is easily illustrated by counting the number of rules required for expressing distributivity of *,/ over +, -. In the expression ***a*b***, if ***b*** is an addition/subtraction expression, then ***a*** can be distributed

and vice-versa. ***b*** can either be an addition/subtraction operation. Also, ***** can be replace by ***/***. Therefore, the number of rules required = 2*2*2 = 8. Through this example, one can see how the number of semantic rules required may grow exponentially.

Most often, the non-terminals of a CFG, do not directly represent domain specific constructs. Sometimes, grammars are altered to fit the requirements of LL(k) or LR(k) parsing. This modification requires additional non-terminals without any meaning in the domain. For instance, the grammar shown in (figure 1) is left recursive and left-factoring has to be performed to convert it into a LL(1) grammar.

E → TE'
E' → +TE' | –TE' | ε
T → FT'
T' → * FT' | / FT' | ε
F → <id> | (E)

Additional non-terminals (T' and E' above), further hamper the expressiveness of the rewrite rules.

Provided these rewrite rules, a backtracking based algorithm can be developed to match the two strings that are semantic equivalent by applying the rewrite rules recursively. This algorithm is naturally exponential in the worst case because at each rule match, two possibilities have to be explored and in the worst case all the nodes in the tree match the rule. The above informal argument clarifies why exponential time is required in the worst case for matching. However, the search space for backtracking can be pruned by using the second string as the guide. One approach to pruning is to use lookahead. If the parse trees have matched up to a certain level, the next level of both trees may be compared to determine which semantic rule to apply. This approach is limited and cannot be applied on all kinds of tree-rewrite rules.

Faster Matching algorithm using Hashing

Exponential complexity is unacceptable for practical applications. To attack this problem from a practical perspective, a randomization approach is considered. A monte carlo algorithm, with one-sided error probability is developed using the idea of hashing. The idea is to develop a hash function, to hash parse trees and match the hash values. If the hash values do not match, then the strings are not semantically equal. If the hash values match, depending on the quality of hash function, the probability that the strings are semantically equal is very high. Hash collisions cannot be avoided. The semantic rules are encoded in the hash function. It must be ensured that the calculation of hash value does not take exponential time.

A Hash function for tree structures can be defined recursively. It must capture the parent-child relationship, branching factor and semantics. The hash value of a node is a function (***Mixing*** function) of its content and the hash values of its children. Nature of mixing function affects the semantics captured by the hash function. For example, the tree rewrite

rule for commutativity of addition can be captured by using a commutative mixing function (like XOR). Encoding associativity will require us to examine more than one level in the parse tree as the operands involved in an associative operation span over more than one level. To encode associativity, depth of each node must be used in the hash function. By treating operands of an associative operator at the same level, associativity can be encoded. This is similar to the syntax directed translation phase (AST creation) phase in compiler construction.

Pre-processing Step

An optional pre-processing step to convert the parse tree into canonical form may be used for two reasons :

(1) Semantic rules may be easier to express in terms of the canonical form.
(2) Elimination of additional nonterminals introduced while converting grammar to LL(k) or LR(k).

Parsing and conversion to canonical parse tree may be done in a single step if canonicalization can be expressed as an syntax directed translation. Converting to canonical form may not be computationally feasible in all cases. For example, converting boolean expressions to CNF is exponential in the worst case.

Implementation for XML Semantics

XML Schema provides both syntax and semantics definitions for matching XML documents. Schema provides three ways to express order among the elements : ***choice***, ***sequence***, and ***all***. *Choice* is a selector between multiple rules, *sequence* specifies an ordered list of elements and *all* specifies an unordered list of elements. Nesting of *all* withing *choice/selector* and vice-versa is not allowed. XML attributes are always unordered. It is assumed that two XML documents are semantically equal if they only differ in the order of elements in an unordered derivation.

This was tested using the JAVAX DOM Parser. A single pass over the schema file is sufficient to determine if an element has ordered/unordered children. This information is aggregated into a hash table. The hash function uses commutative/non-commutative hash as per the ordered/unordered flag status of an element.

Interactive XML Schema Creator

To aid domain experts in specifying semantic rules through XML Schema, an application software providing an interactive UI to create XML Schema is designed. The design of the application is described below:

UI

The top-level window is split (vertically) into two panes. The left pane is used to manage the tree structure. The right pane is used to set properties of nodes in the trees.

Left Pane

Displays the tree structure in the form of foldable nodes.

- Nodes are selectable.
- Nodes can be re-ordered, moved up/down the hierarchy using the drag-and-drop interface.
- When a node is double clicked, a context menu is opened in right pane. (details provided in the next section).

Adding new nodes to the hierarchy:

1. Select a parent node (by single click) and click the add button (placed in the status bar in left pane). A new node will be added as the last child. Properties of the node (including name) can be edited using right pane.
2. Right click on a node to open a menu with options to add a node above/below the selected node.

A node can be marked as an “Choice” node. This will be a dummy node which behaves just like the XML “Choice” selector. When a node is marked as a choice node, any properties marked in Pane 2 is ignored.

Nodes can be rearranged using drag-and-drop.

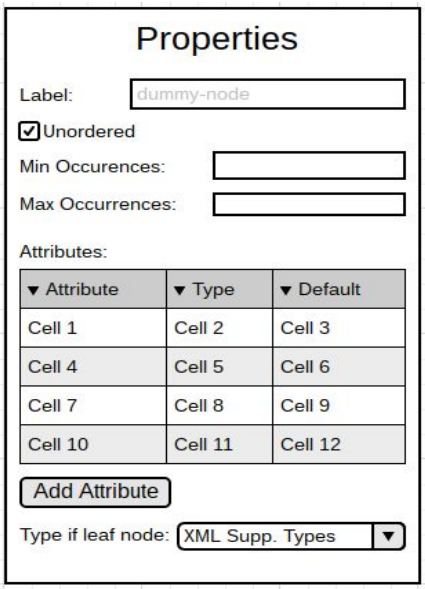
Right Pane

Interface to manage properties of a node.

- Element label

- Ordered/Unordered
- Num Occurrences
- Attributes
- Type if leaf node

The above properties can be modified using the interface.



Implementation

The program can be implemented using the HTML, CSS and JS web frontend technologies. A new platform called “Electron” based on NodeJS and Chrome V8 JS engine for supporting development of desktop applications using HTML, CSS
and JS is available as free & open source software. This platform can be utilized to create responsive and asynchronous UI using Jquery related packages. The tree structure can be displayed using the “Jqtree” [3] package. Jqtree supports event listeners for actions such as “nodeSelected”, “nodeDoubleClicked”, “nodeDragged”, etc.

An intermediate representation is not required as the event listeners can be used to directly manipulate the XML DOM object in the JS. DOM object manipulation provides getNextChild(), getPrevChild(), getParent(), getChildAfter(), etc., methods that can be called when drag and drop is used to modify the tree representation.

The properties of the nodes have direct representation in the XML Schema specification –
The attributes can be represented by – <xs:attribute> tag. The default value can be specified in the same tag. The choice nodes can be specified using the <xs:choice> tag. The nesting in the tree has to be translated to nesting of XML tags. Children of unordered nodes can be enclosed in <xs:all> tag and ordered nodes can be enclosed in <xs:sequence> tag. The element tag has minOccurance and maxOccurance attributes that can be inserted from the input.

The UI also has to generate an interface to input data interactively to output XML. The generated interface will be similar to the above UI except that the tree structure will be locked for modification and the data for leaf nodes can be filled using the right pane.

Conclusion and Future Work

The project provides an efficient scheme to semantically match strings generated by a context free grammar, demonstrates the approach by applying it to XML domain and a design to support domain experts by UI is illustrated.

Further extensions to the project can be in the following directions -
[1] Chalk out a design process/algorithm to translate tree-rewrite rules into hash functions.
[2] Argue and Establish the theoretical bounds for error probability of the designed hash functions.
[3] Consider a larger class of semantic rules for development of hash functions. XML Schema has limited expressiveness to specify semantic rules.
[4] Apply/Extend the process for unstructured data.

References

[1] *Coccinelle,* Inria Labs, [Computer Software]
http://coccinelle.lip6.fr/

[2] Baxter, I. D., Yahin, A., Moura, L., Anna, M. S., & Bier, L. (1998, November). Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on* (pp. 368-377). IEEE.

[3] XML Reference, W3 Consortium [Whitepaper]
https://www.w3.org/XML/

[4] XML Schema Reference, W3 Consortium [Whitepaper]
https://www.w3.org/XML/Schema

[5] Hash functions

Knuth, D. E. (1998). *The art of computer programming: sorting and searching*(Vol. 3). Pearson Education.

Thomas H.. Cormen, Leiserson, C. E., Rivest, R. L., & Stein, C. (2001).*Introduction to algorithms* (Vol. 6). Cambridge: MIT press.

Bob Jenkins (1997), *Hash functions*. Dr. Dobbs Journal, September 1997.