A TERSE STRING-EMBEDDED LANGUAGE FOR TREE SEARCHING AND REPLACING

Matúš SULÍR*, Slavomír ŠIMOŇÁK**
*Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic, e-mail: matus.sulir@student.tuke.sk
**Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic,tel. +421 55 602 3021, e-mail: slavomir.simonak@tuke.sk

ABSTRACT

Treepace is a new library and a domain specific language for tree pattern matching and replacing, implemented in Python. Selected concepts resemble the well-known application programing interface (API) for string regular expressions. The language is terse since it is possible to write a simple transformation consisting of a pattern and a replacement in one row. Objects of any types can be used as node values. Calling host language constructs, e.g., functions, in the embedded language is straightforward. Node class inheritance allows for mapping tree nodes to external objects like GUI (graphical user interface) components.

Keywords: *tree transformation, application programming interface, domain-specific language, pattern matching, string embedding*

1. INTRODUCTION

Many applications operate on tree-like data structures, consisting of nodes and edges. Compilers and other computer language processing applications use abstract syntax trees (ASTs) as one of intermediate representations of programs. Data processing applications convert information from semantic to visual representations, e.g., an address book containing items to an HTML document. More examples can be found in [1].

In this paper, we consider all tress rooted (one node is declared a root), node-labeled (each node can be assigned a value) and ordered – the order of nodes is important.

1.1. The purpose

The purpose of the library Treepace, presented in this paper, is to extract parts of trees according to a pattern and optionally replace them by another subtrees.

Examples of effects possible to achieve with Treepace include: mathematical expression simplification and solving, XML to HTML transformation, execution of a simple program in a form of an AST, etc.

1.2. Existing XML solutions

First, we will focus on languages operating on XML documents – which are inherently tree structured. XPath [2] is one of the most used existing languages to address tree parts. An XPath query selects a set of nodes from the tree according to a series of location steps [3]. In essence, each location step contains an axis determining the searching direction (e.g., children, following siblings) and conditions which the searched nodes must meet [4]. The resulting node set corresponds to the last location step. For instance, the following XPath expression selects all headings of an article saved in a form of an XML document: `/child:article/descendant:heading`.

While XPath is a succinct and expressive language, it does not support the modification of trees. There exist languages like XSLT and XQuery Update for this purpose. The former does not fully preserve the simplicity and terseness of XPath. It is based on rules; each rule consists of a pattern and a template used to build a new tree [5]. An excerpt from a simple transformation changing headings to HTML "H1" tags follows:

```
<xsl:template match="/article//heading">
    <h1><xsl:apply-templates/></h1>
</xsl:template>
```

XQuery Update [6] is a bit more succinct, but some shortcomings of XML persist. Because XML documents must be serializable to strings, their node values cannot be arbitrary objects (e.g., open file handles or network sockets). In addition, it is difficult to react on node changes because the only output of a transformation is a resulting tree.

1.3. Other solutions

Two more candidates are not general enough to be suitable for all mentioned uses. JastAdd, a system based on Rewritable reference attributed grammars [7] is focused on AST transformations. Tsurgeon is specialized for linguistic applications [8].

A tree is a special case of a graph, so it is worth mentioning a graph manipulation language Gremlin [9]. It provides a succinct way of graph traversal and data filtering through a series of steps, utilizing the Java Virtual Machine. However, it operates on a graph database, so it is more suitable for persistent data manipulation and less for an AST transformation or document transformation.

2. THE APPLICATION PROGRAMMING INTERFACE

Treepace is designed as a domain-specific language embedded in Python, so it offers the programmer a convenient Python API.

The basic building block of trees in Treepace is an object of class `Node`. A tree (of class `Tree`) is defined by a reference to the root node. When matching against the tree, subtrees (`Subtree`) are returned. Each subtree is defined by a set of node references pointing to the main tree.

Table 1 An analogy between regex and Treepace API calls

Action	Python regex API call example	Treepace API call example
search anywhere	`re.search('pattern', text)`	`tree.search('pattern')[0]`
search from the start (root)	`re.match('pattern', text)`	`tree.match('pattern')`
match the whole text / tree	`re.fullmatch('pattern', text)`	`tree.fullmatch('pattern')`
replace a part of the text / tree	`re.sub('pattern', 'replacement', text)`	`tree.replace('pattern', 'replacement')`
transform a tree (replace using rules in a loop while matches are found)	–	`tree.transform(''' pattern1 -> replacement1 pattern2 -> replacement2 ''')`

Trees can be manually constructed or loaded from a format like XML, parenthesized or indented text. After a tree is loaded in the memory, we can perform operations like searching an replacing. The API of Treepace library was designed to be conceptually similar to the standard Python regular expression library. The comparison is in Table 1.

Like regex, our language is string-embedded [10] – the embedded language (Treepace) programs are written as strings in the host language (Python) programs. Patterns, replacements and rules are passed as arguments to the API methods. However, Treepace modifies the original tree while a regular expression replacement returns a new string.

Values of nodes in our library can be of any type – from simple strings and numbers through structured data (e.g., associative arrays) to complicated objects bound to external entities like open file descriptors or computer display areas. Thus, the transformations can perform actions on these objects according to patterns – for example, delete all files matching a condition.

It is possible to inherit from the basic node class. This is useful to add various side effects to each node change. In Treepace, there are two examples included – a logging node and a GUI node (see Fig. 1 for a class diagram). The former writes an information about each changed node to the standard output. The latter displays a visualization of the transformation process in a GUI window – see Fig. 2. Every time a node is changed, the corresponding GUI component part is updated. We can say that each node in the graphical component is mapped to a node in the memory. Every transformation consists of a sequence of these elementary operations:

- Create a node.
- Change the value of a node.
- Insert a node at the given position of the child list of a particular node.
- Detach a node from its parent.

Each class inherited from `Node` has to override just the mentioned methods to achieve the mapping.

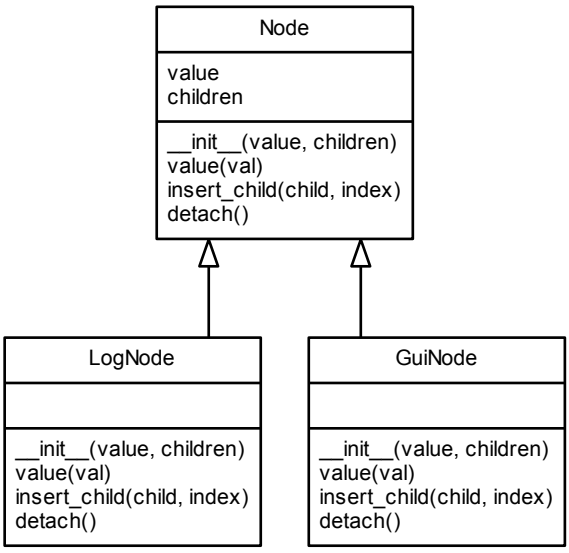


Fig. 1 Node class inheritance, adding side effects to tree transformation

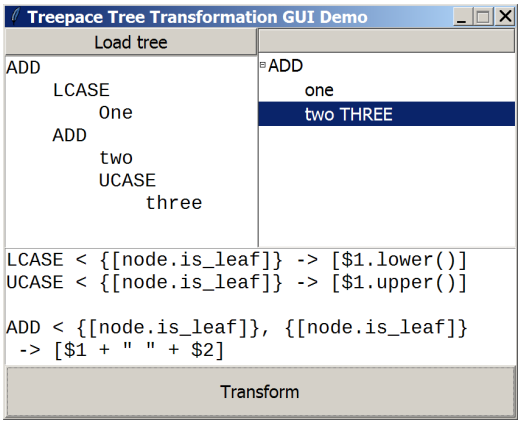


Fig. 2 GUI-mapped tree nodes during transformation

3. THE TRANSFORMATION LANGUAGE

For the reference, the transformation language syntax will be expressed in a form of a PEG (Parsing Expression Grammars) grammar. In contrast to generative context-free grammars, PEGs are recognition-based. They incorporate both the lexical and syntactical grammar in one unit. Alternatives, traditionally denoted by "|", are replaced by ordered alternatives – "/", so they are always unambiguous. More details about PEGs can be found in [11].

The transformation language grammar is located in Fig. 3. The starting rule *Rule* represents a Treepace transformation rule and divides it into two logical parts – a pattern and a replacement. The rule "_" allows whitespace characters to be present at convenient locations. Now we will describe the components of the language.

```
      Rule ← Pattern '->' Replacement
         _ ← (' ' / '\t')*

   Pattern ← Group (RelGroup)*
     Group ← Node/(GroupStart Pattern GroupEnd)
  RelGroup ← (Relation Group) / ParentAny
      Node ← Any / Constant / Code / Reference
       Any ← _ '.' _
  Constant ← _ (('\w'+) / ('"' (!'"' .)+ '"')) _
      Code ← _ '[' PythonCode ']' _
PythonCode ← ExprPart+
  ExprPart ← ( !('[' / ']') .)+ / ('[' ExprPart ']')
 Reference ← _ '$' RefNum _
    RefNum ← '\d'+
GroupStart ← _ '{' _
  GroupEnd ← _ '}' _
  Relation ← Child / Sibling / NextSibling
     Child ← _ '<' _
   Sibling ← _ '&' _
NextSibling ← _ ',' _
 ParentAny ← _ '>' _

Replacement ← ReplNode (ReplRelNode)*
   ReplNode ← Constant / Code / Reference
ReplRelNode ← (ReplRelation Node) / ParentAny
ReplRelation ← Child / NextSibling
```

Fig. 3 The Treepace transformation language PEG grammar

3.1. One-node patterns

The pattern, represented by the rule *Pattern* in Fig. 3, is used for searching in a tree, both standalone and as a part of replacing or transformation. At first, we will focus on node tests – simple patterns matching one node.

The most basic pattern, matching one arbitrary node (*Any*) is a dot (`.`). A literal text (*Constant*) searches for a node whose string representation is equal to it, e.g., a pattern `lit1` matches any node labeled "lit1".

The most powerful one-node test is a predicate (*Code*) enclosed in square brackets, which can contain any Python expression. A node matches the given predicate if the expression evaluates to `True`. The code is evaluated in an environment containing a reference to the currently tested node (the variable `node`). For example, the predicate `[node.value != "x"]` matches all nodes whose value does not equal "x". An underscore is a shortcut for `node.value`. For instance, `[_.upper() == "Z"]` matches all nodes whose uppercase string representation equals "Z".

Auxiliary variables can be supplied to the code via keyword arguments of the API methods. This method call returns a list of all one-node subtrees containing a node whose value is less than or equal to 7.4: `tree.search('[_ <= limit]', limit=7.4)`.

3.2. Relations

We can combine multiple node tests into a pattern using relations (*Relation*). The pattern has essentially the form

$$t_1 \quad R_{1,2} \quad t_2 \quad R_{2,3} \quad t_3 \quad \ldots \quad R_{n-1,n} \quad t_n$$

where t_i and t_j are node tests and $R_{i,j}$ is a relation between a node matching the test t_i and a node fulfilling the test t_j.

The relation "to have a child" is denoted by "`<`". Thus, the pattern `a < b < c` matches subtrees with the root labeled "a", having a child "b" which has a child "c".

Other possible relations are "to have a sibling" (`&`), an immediately following sibling (`,`) and a parent: `>`. The parent relation is implicitly followed by a node test matching any node. For example, the pattern `root < a <b>, c & d` matches the highlighted subtree in Fig. 4.

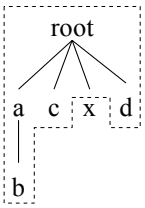


Fig. 4 An example of a matched subtree

3.3. Groups

Parts of a pattern can be enclosed in braces to form a group (*Group*). The groups are numbered from one and can be nested. For instance, in the pattern `{n1 < {n2}}`, the subtree containing nodes n1 and n2 is saved as the group number one; the second group contains the node n2. A subtree matching the whole pattern implicitly becomes a group number 0.

It is possible to back-reference a group (*Reference*) both in the pattern and a replacement using the notation `$n` where `n` is the group number. This behavior strengthens the similarity between Treepace and regular expression implementations in many general-purpose languages. For example, the pattern `{.} < $1` matches two nodes in a parent-child relationship, having equal values.

3.4. The replacement

The replacement part (*Replacement*) of a rule consists of nodes separated by relations which they should form. Each node in the replacement can be a literal, a back-reference or an arbitrary Python expression. In the last case, the code is evaluated and its result is assigned to the node value. Supported relations are: child, next sibling and parent. For instance, the replacement string `[abs(-1)] < $2` represents a tree with the root labeled "1", having a subtree number 2 connected as its child.

A found subtree is intuitively replaced by a replacement tree according to the strategies described in the section 4.4.

3.5. Rules

Combining a pattern and a replacement, we get a rule (*Rule*) in the form: `pattern -> replacement`. To give an example, the rule `range < {.}, {[$1 > _]} -> range < $2, $1` swaps the values of two children of a node labeled "range" if the value of the first children is greater than the second.

4. THE EXECUTION

First, the pattern and replacement is parsed into a syntax tree according to the grammar (Fig. 3) and converted to two lists of instructions – one for searching and one for building a replacement of a particular subtree. Matches are found by executing the searching instructions. If the matches overlap, an exception is raised, otherwise the replacing could produce an unspecified behavior. For each matched subtree, a replacement is built by executing the tree-building instructions and the original subtree is replaced by a new one.

4.1. The instruction generation

The syntax tree is traversed in a postorder manner. When a particular node is visited, the corresponding instruction is generated and appended to the list of instructions. The mapping from nodes to instructions is shown in Fig. 5. The scheme $\mathscr{T}_S$ is used for the pattern part of the syntax tree, $\mathscr{T}_R$ for the replacement part. Unmentioned node types do not produce any instructions. If there is a symbol in angle brackets following the node type, its value is a part of the input string corresponding to the given node. The function `start_num()` returns the number of a currently starting group, `end_num()` the number of an ending group.

4.2. Searching

The searching instructions are executed on a tree-searching virtual machine which consists of a list of searching branches. Each branch is a quintuple:

$$(groups, match, node, rel, instrs)$$

Every searching branch can be understood as a candidate for the match accompanied by the necessary state information. The set *groups* contains numbers of groups we are currently enclosed by in the pattern. It is initialized to $\{0\}$. The list *match* consists of a matching subtree (group 0) and optionally other subtrees (one for each group). It initially contains one empty subtree. *Node* is the current context node, initialized to a root node. The item *rel* represents a relation we are currently using for searching. Its initial value is id (identity) for `match()` and `fullmatch()` methods; descendant (descendant or the node itself) otherwise.

The execution algorithm is as follows: For each branch whose instruction list *instrs* is nonempty, an instruction is popped from the beginning of the list and executed on the given branch with the following effect:

- FIND p replaces the current branch by zero or more branches, one for each node which is in the relationship *rel* with the node *node* and matches the predicate p.
- REL r sets *rel* to r.
- GRPS n represents a start of the group n. It adds n to the set *groups* and an empty subtree to the list *match*.
- The group end – GRPE n – removes n from *groups*.
- REF n (the back-reference) generates a list of instructions which searches for a subtree equal to the subtree saved in $match[n]$. It adds the generated list to the beginning of *instrs*.

The process repeats until there are no branches with a nonempty instruction list. After execution, the resulting subtrees are in the *match* fields of the branches.

$$\begin{aligned}
\mathscr{T}_S[\![Any]\!] &= \boxed{\text{FIND 'True'}} \\
\mathscr{T}_S[\![Constant\langle c\rangle]\!] &= \boxed{\text{FIND 'str(node.value)==str(}c\text{)'}} \\
\mathscr{T}_S[\![PythonCode\langle c\rangle]\!] &= \boxed{\text{FIND } c} \\
\mathscr{T}_S[\![RefNum\langle n\rangle]\!] &= \boxed{\text{REF } n} \\
\mathscr{T}_S[\![GroupStart]\!] &= \boxed{\texttt{GRPS start_num()}} \\
\mathscr{T}_S[\![GroupEnd]\!] &= \boxed{\texttt{GRPE end_num()}} \\
\mathscr{T}_S[\![Child]\!] &= \boxed{\text{REL child}} \\
\mathscr{T}_S[\![Sibling]\!] &= \boxed{\text{REL sibling}} \\
\mathscr{T}_S[\![NextSibling]\!] &= \boxed{\text{REL next_sibling}} \\
\mathscr{T}_S[\![ParentAny]\!] &= \boxed{\begin{array}{l}\text{REL parent}\\ \text{FIND 'True'}\end{array}} \\
\\
\mathscr{T}_R[\![Constant\langle c\rangle]\!] &= \boxed{\texttt{ADD repr(}c\texttt{)}} \\
\mathscr{T}_R[\![PythonCode\langle c\rangle]\!] &= \boxed{\text{ADD } c} \\
\mathscr{T}_R[\![RefNum\langle n\rangle]\!] &= \boxed{\text{AREF } n} \\
\mathscr{T}_R[\![Child]\!] &= \boxed{\text{AREL child}} \\
\mathscr{T}_R[\![NextSibling]\!] &= \boxed{\text{AREL next_sibling}} \\
\mathscr{T}_R[\![ParentAny]\!] &= \boxed{\text{GPAR}}
\end{aligned}$$

Fig. 5 The translation schemes

4.3. Building a replacement

For each single match *match*, a replacement building machine is created:

$(match, instrs, node, rel, tree)$

where *instrs* are instructions obtained from the scheme $\mathscr{T}_R$, *node* represents the current context node, *rel* is the relation used to add the next node (child or next_sibling) and *tree* contains a reference to the root node of the tree being built. Instructions are popped from the front of the list *instrs* and executed:

- ADD *c* creates a new node *n* with a value equal to the execution result of the Python code *c*. The node *n* is added to the tree in such way that *node* and *n* form a relation *rel*. The context *node* is set to *n*.
- AREL *r* assigns *r* to *rel*.
- The back-reference AREF *n* adds the subtree $match[n]$ to the tree in a way that the node *node* and the subtree's root form a relation *rel*. The context *node* is set to the subtree root.
- GPAR sets *node* to its parent.

The replacement is then stored in the field *tree*.

4.4. Replacing

Finally, each matched subtree is substituted by its replacement. Because trees are not linear structures like strings, not every subtree can by replaced by any other tree. For example, we can not intuitively determine the resulting tree when a one-node subtree having two children in the main tree is to be replaced by a tree with three children (Fig. 6). Enumerating all ambiguous cases would be difficult (if even possible), so we decided to define a list of five intuitive and unambiguous replacing strategies. Each of them contains a test whether it can be applied and the application algorithm itself. The strategies are tried from the highest priority to the lowest one. If no strategy can be applied, an exception is raised.

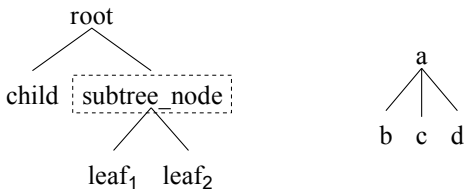


Fig. 6 A tree with a found subtree (left) and its ambiguous replacement (right).

The highest priority strategy is applicable if a match and a replacement are of the same shape. Its algorithm is to assign each node value of the old subtree to the corresponding node of the new tree (Fig. 7).

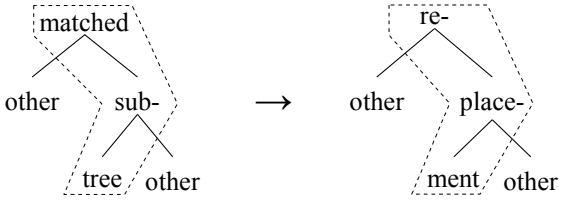


Fig. 7 The same tree shapes strategy

The second strategy replaces an arbitrary subtree by one node. An example of such replacement is in Fig. 8.

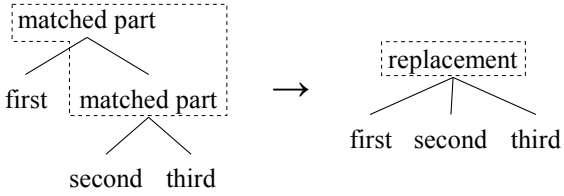


Fig. 8 Replacing a subtree by one node

If a matched subtree does not have any children in the main tree, it is replaced by the third strategy (Fig. 9). The match can be considered one big leaf which is replaced as a whole.

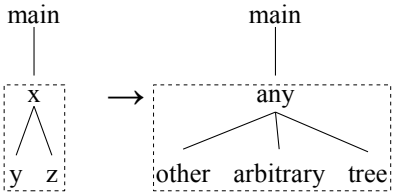


Fig. 9 A subtree with no children in the main tree

The last two strategies require that the matched subtree's inner nodes (including the root) must not have children outside the subtree. If the match have the same leaf count as the replacement, children of the match leaves become the children of the replacement leaves (Fig. 10).

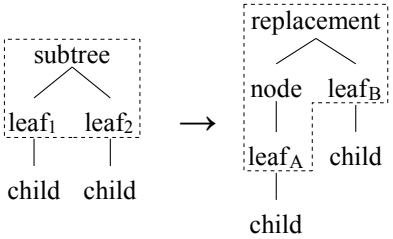


Fig. 10 The same leaf count

Leaves of a subtree which have children in the main tree will be called "partial leaves". If the partial leaf count of a

match equals the leaf count of a replacement, we can use the last strategy (Fig. 11).

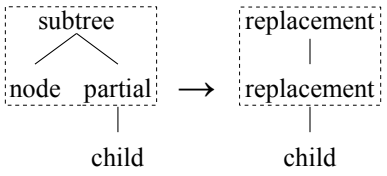
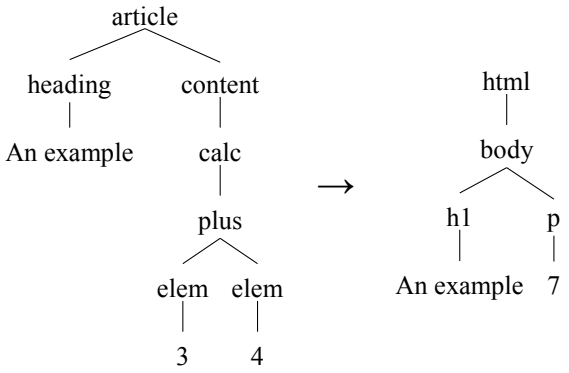


Fig. 11 The same partial leaf count

5. EXAMPLES

5.1. A document transformation

Fig. 12 displays the structure of a tree extracted from an XML document and a resulting tree after execution of the transformation written in Treepace. The practical expressivity of the language is high in comparison to existing languages. The exhibited source code has a length of around 40% (measured by character count) compared to the same transformation written in XQuery Update and only 30% of the length of XSLT code.




```
1 tree.transform(’’’
2    article -> html < body
3    heading -> h1
4    content -> p
5    calc < plus < elem<{.}>, elem<{.} ->
         [text(num($1) + num($2))]
6 ’’’)
```

Fig. 12 An example of a transformation and its source code

5.2. A transformation execution example

We will follow execution of a sample transformation `tree.transform('["a" in _] < {.} -> $1 < x')` on the tree displayed on the left side of Fig. 13. First, the rule is parsed into a syntax tree. Its relevant parts (with some nodes omitted) are shown in Fig. 14.

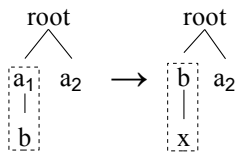


Fig. 13 The tree before transformation → after transformation

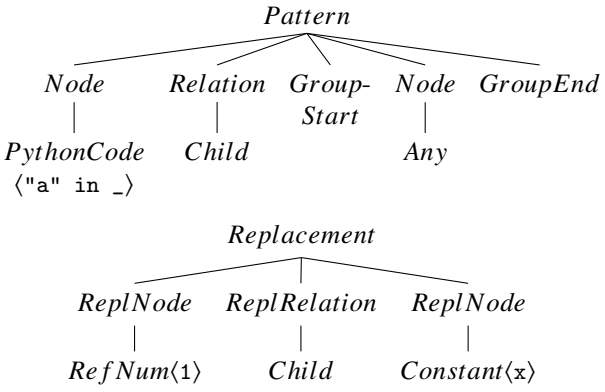


Fig. 14 Parts of an abstract syntax tree obtained from the rule

The pattern part of the rule produces the instruction list:

[FIND '"a" in _', REL child, GRPS 1, FIND True, GRPE 1]

For the replacement part, the generated instructions are:

[AREF 1, REL child, ADD '"x"']

Searching is performed on the searching machine – a list of branches (*groups*, *match*, *node*, *rel*, *instrs*). The sequence of machine states is as follows:

$$\big[(\{0\}, [\{\}], \text{root}, \text{descendant}, [\text{FIND '"a" in \_'}, \ldots])\big] \Rightarrow^{(1)}$$
$$\big[(\{0\}, [\{\underline{a_1}\}], \underline{a_1}, \text{descendant}, [\text{REL child}, \ldots]),$$
$$(\{0\}, [\{\underline{a_2}\}], \underline{a_2}, \text{descendant}, [\text{REL child}, \ldots])\big] \Rightarrow^{(2)}$$
$$\big[(\{0\}, [\{a_1\}], a_1, \underline{\text{child}}, [\text{GRPS 1}, \ldots]),$$
$$(\{0\}, [\{a_2\}], a_2, \underline{\text{child}}, [\text{GRPS 1}, \ldots])\big] \Rightarrow^{(3)}$$
$$\big[(\{0,\underline{1}\}, [\{a_1\}, \underline{\{\}}], a_1, \text{child}, [\text{FIND 'True'}, \ldots]),$$
$$(\{0,\underline{1}\}, [\{a_2\}, \underline{\{\}}], a_2, \text{child}, [\text{FIND 'True'}, \ldots])\big] \Rightarrow^{(4)}$$
$$\big[(\{0,1\}, [\{a_1,\underline{b}\}, \{\underline{b}\}], \underline{b}, \text{child}, [\text{GRPE 1}])\big] \Rightarrow^{(5)}$$
$$\big[(\{0\}, [\{a_1,b\}, \{b\}], b, \text{child}, [])\big]$$

The first instruction (1) searches for all descendants (including itself) of the root node whose label contains the letter “a”. There are two such nodes, so the current searching branch is divided into two, having the corresponding group 0 of each *match* updated and the context *node* set. A relation-setting instruction is executed (2) on each of the branches. A start of a new group (3) adds the group number to the current group set and appends a new, empty subtree to the match. Then all children of the context nodes are found (4). Since the context node of the second branch, a_2, does not have a child, this branch is removed from the machine.

After the group number 1 ends (5), the instruction list of the sole remaining branch is empty and the search finishes.

For each match (in this case, only one) a replacement-building machine $(match, instrs, node, rel, tree)$ is created. The execution process follows.

$$([\{a_1,b\},\{b\}],\ [\texttt{AREF 1},\ldots],\ None,\ None,\ None) \Rightarrow^{(1)}$$
$$([\{a_1,b\},\{b\}],\ [\texttt{REL child},\ldots],\ \underline{b},\ None,\ \underline{b}) \Rightarrow^{(2)}$$
$$([\{a_1,b\},\{b\}],\ [\texttt{ADD '"x"'}],\ b,\ \underline{child},\ b) \Rightarrow^{(3)}$$
$$([\{a_1,b\},\{b\}],\ [\,],\ \underline{x},\ child,\ \begin{matrix}b\\|\\\underline{x}\end{matrix})$$

The back-referencing instruction (1) creates a new tree (because it is not yet created) containing the subtree from the group 1, which is the sole node b. It sets the context node to the root of the added tree – b. Then a relation is set (2) and a node with the value of "x" is added to the tree (3) as a child of the context node.

The matches are not overlapping (since there is only one match) and the replacement can proceed. Because the pattern and the replacement are of the same shape (two nodes – a parent and a child), the first strategy is used. The value of node a_1 is replaced by "b" and the value of node b by "x". The result is shown on the right side of Fig. 13.

The transformation then stops because the search pattern is no longer present in the tree.

6. CONCLUSION

Our goal was to create a terse, string-embedded language for tree transformations where the node values can be arbitrary objects and nodes can react to changes which occur during transformation. The language is not limited to a particular application domain like natural language processing or AST transformation. Functionalities of the host language can be accessed easily within the embedded code. The ability to react on node changes has an advantage that tree-resource mapping like a simple tree visualization can be implemented with minimum effort.

To lower the barrier of learning a new language, several concepts resemble popular string regular expression implementations to some extent. Furthermore, an online tutorial using the IPython Notebook [12] technology is available at `http://nbviewer.ipython.org/github/sulir/treepace/blob/master/doc/Tutorial.ipynb`.

A short comparison of selected features with existing similar languages can be found in Table 2.

6.1. The language expressivity

The expressivity of Treepace patterns is comparable to XPath. The concept of relations in our solution is similar to XPath axes. An important difference is that Treepace does not support the descendant (a direct or indirect child) relation. This is justified since XPath queries return only a set of nodes while Treepace matches a continuous subtree.

The theoretical expressivity of Treepace is limited by the fact its current version does not support any form of a repetition or alternative in the pattern. Hence for the given pattern string, the matched subtree has a fixed number of nodes.

Table 2 A comparison of Treepace with similar languages

	terseness	arbitrary objects as node values	embedding type
Treepace	yes	yes	string
XSLT	no	no	none
XQuery Update	partial	no	various
Tsurgeon	yes	no	none
JastAdd	no	yes	none
Gremlin	partial	no	pure, string

Regarding the replacement, there are two main limiting factors. First, the matches must not overlap. Second, there are currently only five replacing strategies built-in. However, it is possible to use only the pattern matching part of the library and perform the replacement manually.

6.2. Possible extensions

It would be interesting to implement patterns similar to Regular XPath [13], particularly the transitive closure support. This would allow using patterns like `root < child (,child)*`. The star in the pattern would represent a repetition, so it would select all direct children (labeled "child") of the root node.

Often used concepts like checking whether a node is a leaf, should have their own syntactic sugar. Thanks to the object-oriented nature of the library source code, it will be easy to add additional relations or replacing strategies to Treepace.

More details about Treepace can be found in the master's thesis [14].

ACKNOWLEDGEMENT

This work was supported by VEGA Grant No. 1/0341/13 Principles and methods of automated abstraction of computer languages and software development based on the semantic enrichment caused by communication.

REFERENCES

[1] SULÍR, M. – ŠIMOŇÁK, S.: Methods and Application of Tree Transformations. Electrical Engineering and Informatics IV: Proceedings of the Faculty of Electrical Engineering and Informatics of the Technical University of Košice, 2013. 373–377

[2] CLARK, J. et al.: XML Path Language (XPath). 1999. `http://www.w3.org/TR/xpath`

[3] GENEVÈS, P.: Course: The XPath Language. Grenoble: University of Grenoble, 2014. `http://wam.inrialpes.fr/courses/PG-MoSIG13/xpath.pdf`

[4] GOTTLOB, G. – KOCH, C. – PICHLER, R.: *XPath processing in a nutshell*, ACM SIGMOD Record **32**, No. 2 (2003) 21–27

[5] CLARK, J. et al.: XSL Transformations (XSLT). 1999. http://www.w3.org/TR/xslt

[6] ROBIE, J. et al.: XQuery update facility 1.0. 2011. http://www.w3.org/TR/xquery-update-10/

[7] EKMAN, T. – HEDIN, G.: Rewritable reference attributed grammars. ECOOP 2004 – Object-Oriented Programming. 147–171

[8] LEVY, R. – ANDREW, G.: Tregex and Tsurgeon: tools for querying and manipulating tree data structures. Proceedings of the fifth international conference on Language Resources and Evaluation, 2006. 2231–2234

[9] MILLER, J. J.: Graph Database Applications and Concepts with Neo4j. Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA. 2013.

[10] ERDWEG, S. – GIARRUSSO, P. G. – RENDEL, T.: Language composition untangled. Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications. 2012

[11] FORD, B.: *Parsing expression grammars: a recognition-based syntactic foundation*, ACM SIGPLAN Notices **39**, No. 1 (2004) 111–122

[12] PEREZ, F. – GRANGER, B. E.: An Open Source Framework for Interactive, Collaborative and Reproducible Scientific Computing and Education. 2013. http://ipython.org/_static/sloangrant/sloan-grant.pdf

[13] TEN CATE, B.: The expressivity of XPath with transitive closure. Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. 2006. 328–337

[14] SULÍR, M.: Language of tree transformation design and implementation. Master's thesis. Košice: Technical University of Košice. 2014 (in Slovak)

Received May 26, 2014, accepted June 22, 2014

BIOGRAPHIES

Matúš Sulír was born in Poprad, Slovakia in 1991. In 2014 he graduated with a master's degree in Informatics from the Faculty of Electrical Engineering and Informatics at Technical University of Košice. Currently he continues his studies at the doctoral level. His research interests encompass domain-specific languages, programming paradigms, computer emulation and web technologies.

Slavomír Šimoňák was born in 1974. In 1998 he graduated from the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics at Technical University of Košice and defended his PhD thesis titled "Formal Method Integration Based on Transformation of Petri Nets and Process Algebras" in 2003. Currently he works as an assistant professor at the Department of Computers and Informatics. His research interests include formal methods integration and application, algorithms and data structures, machine-oriented languages, and computer emulation.