Université Paris-Saclay
Master 2 Data and Knowledge

Paris, November 18, 2017
*Professor: Silviu Maniu*

# Web Data Models - Project

Paweł Guzewicz[1]

## Implementation

The aim of the project is to program the validator of XML files with respect to simple DTD files. I implemented it in C++ and tested using Python. Code is divided into 4 modules: `main`, `XML_Parser`, `XML_Tree` and `XML_DTD_Validator`.
The algorithm for evaluation of the validity of XML file, given DTD file is performing following computations:

a) Parsing input XML file combined with online check for well-formedness using stack.
b) Building the XML_Tree tree on-the-fly during the parsing that is stored in the memory and will be further traversed during validation against input DTD file. The tree is corresponding to the standard XML document tree. There is a node reserved for the root of the document (that has only one child) that points to the first (unique) element. Then every such node in the tree contains the list of the children (pointers to the child nodes), its name and pointer to the parent for simplifying the navigation (used to ease the construction of the tree, corresponding methods: `add_child` and `go_to_parent`). The tree contains also the context node used during the construction as a reference point and after the construction to double check the well-formedness: we must navigate back to root after processing the whole XML file (also the stack used for the parsing has to be empty).
c) Having checked the well-formedness, the automata corresponding to each line of DTD are declared (optimisation: they will be materialized on demand, only if there is a need for using particular rule; they are built only once, then stored in memory)
d) Checking the validity of the XML file with respect to DTD file is done using depth first search approach (`Tree_node::is_valid`). Each node of XML Tree is checked for match of the list of the children against the regular expression for its name.
e) Matching the regular expression is divided into subprocedures. First for every not constructed automaton, it is build based on regular expression converted to Reverse Polish Notation form using Thompson's construction. It is a non-deterministic automaton (NFA) with epsilon transitions. During it's construction the regular expression is processed placing partial results (partial automata called in my implementation `Automaton_Fragments`) on the stack. Each such fragment consists of two `Automaton_Nodes`, corresponding to the start state and end (final) state of such subautomaton. In every case the final state is pointing to `NULL`, conversely each start state has at least one outgoing transition (lack of transition or epsilon transitions are marked in the special way). Each `Automaton_Node` has 4 fields describing 2 nodes that can be visited using 2 transitions respectively. This means that every state of the NFA can have at most two outgoing labeled transitions. This compact representation is achived thanks to the Thompson's construction.
Reading the regular expression in the postfix form there are 5 possible items that can occur. If the algorithm sees a label, it is translated into automaton with two nodes: start node with one transition to the end node labeled with the label and placed on the stack. If it sees the symbol "." (concatenation), it takes the two top automata from

[1] *E-mail*: `pawel.guzewicz@telecom-paristech.fr`

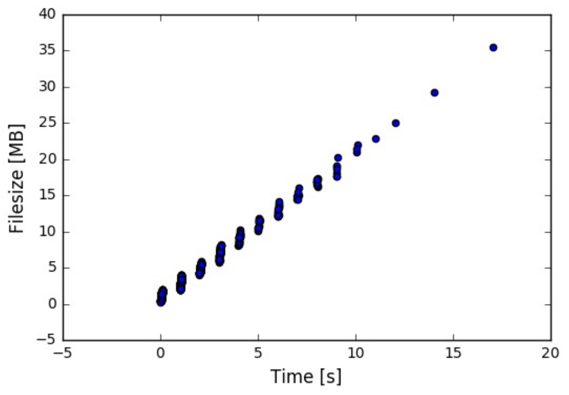
the stack and pushes back one automata that is a concatenation of the two merging end state of the first one with the start state of the second (fragment contains now as a start state a start state of the first and as an end state an end state of the second). For the case of "*" symbol, algorithm takes the automaton from the top of the stack and places back the wrapper of the original automaton with two new nodes added. They correspond to new start and new end node. New start node is linked with the old start node and new end node both with epsilon transitions, the old end node is now pointing to new end node and old start node both with epsilon transitions as well, the new end node is pointing to NULL as usual. There two other possibilities: "?" and "+". It turns out that we need to create exactly the same wrapped automaton as for "*" with the difference from "*" being for "?": dropping the epsilon transition responsible for looping between old end state and old start state and for "+" dropping the epsilon transition from new start state to new end state responsible for possible skipping the label. In the end of the processing we are left with one automaton.

Using such automaton we are able to match the regular expression by simulating non-deterministic execution. Each iteration (each child validated against regular expression) we are computing the set of the possible states (simulation the traversal of all possible paths in NFA). In the end we need to check if the set of states contains the final state. If yes, the node matches regular expression. Each move consists of two phases: we do one step (one transition) using the label from the input (a name of the child) and we do all possible only-epsilon transitions computing so called epsilon-closure. In order to initialize the starting set of states we need to compute the epsilon closure of the root (start state) of automaton.

Programming this validator I made some implementation choices in the spirit of the examples given in the assignment. I assumed that only allowed characters are: one-letter ASCII identifiers (and only alphabetic letter) for the labels of the transitions and ?, +, *, _, ,. Secondly I have taken for granted the validity of the input DTD file as mentioned in the email. I take into account the possible case of two operator (like "?" and "*") following each other and I don't attempt to simplify it nor detect disambiguity, relying of well-formedness of the given DTD. Furthermore I assume that DTD doesn't contain explicit (guaranteed by the assignment specification) nor implicit disjuncions. The former refering to the possible "competition" between the rules in DTD file that might have been interpreted as "or" operation. This leads to assumption that there is one rule per element. Regarding the optimisations, I have already mentioned the lazy construction of the automata. Then I use Thompson's construction which is flattening possible meaningless parentheses thanks to RPN. I also evaluate the results using simple, small NFA simulating its execution in fairly efficient way.

## Experimental evaluation

Experiments invole the execution of the validator on the set of 500 XML documents validated against DTD file. The results of the runs are presented in the figures. They show the linear relation both between the size of the file and the computation time and also between the size of the file and consumed memory. This is expected due to the fact that the whole tree corresponding to the XML file is stored in the memory and because the validation is linear in the size of the parse tree.



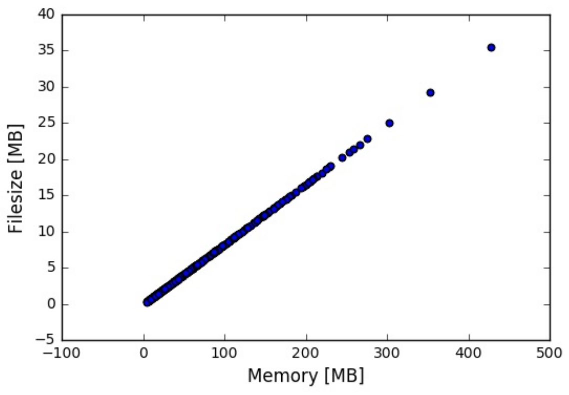


Figure 2: