An Automaton Model for Xcerpt Type Checking and XML Schema Validation

Sacha Berger

January 29^{th} 2007

Abstract

An automaton model used for validation and type checking with languages defined using R_2G_2 [1] is presented. First, tree-shaped data is considered to be handled by the automaton model, then the approach is extended to graph shaped data. The presented approach is based on specialized non-deterministic finite state automata. The specialisation copes with unranked tree shaped data. Graph shaped data will be treated as, possibly inifinite in depth, trees.

The choice of using non-deterministic automata is motivated by complexity issues: as the tree automata are based on regular expressions, non-deterministic automata are a necessary intermediate step. Arguably deterministic tree automata are more eficient on validating data, but the derivation of such automata from non-deterministic ones comes with potentially exponential costs. As all the needed algorithms can be achieved on non-deterministic automata in sub-exponential time and space complexity, no need for determinisation arises.

1 Introduction to Regular Tree Automata

Traditionally, regular tree automata are defined as follows (cf. [2]).

Definition 1 (Non-deterministic Finite Tree Automata) *A non-deterministic finite tree automaton (NFTA) over Σ is a tuple $A = (Q, \Sigma, Q_F, \Delta)$, where Q is a set of (unary) states, $Q_F \subseteq Q$ is a set of final states, and Δ is a set of transition rules of type $f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n))$, where $n \geq 0$, $f \in \Sigma_n$, $q, q_1, \ldots, q_n \in Q$, $x_1, \ldots, x_n \in X$.*

The set Σ contains the symbols or the alphabet of the tree. Note, that traditionally regular tree automata operate on ranked trees, therefore the symbols have fixed arity – the number of child nodes in a corresponding tree is fixed. The set $\Sigma_p \subseteq \Sigma$ is the set of all symbols in Σ with arity p. The set $T(\Sigma)$ denotes the set of all tree that can be constructed using the symbols in Σ. Therefore

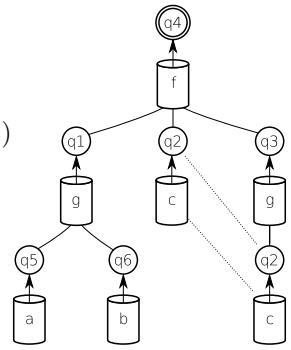
- $\Sigma_0 \subseteq T(\Sigma)$
- for $p \geq 1$, if $f \in \Sigma_p$ and $t_1, \ldots, t_p \in T(\Sigma)$, then $f(t_1, \ldots, t_p) \in T(\Sigma)$

Example 1 *A non-deterministic*[1] *finite tree automaton able to recognize a language containing (under many others) the tree $f(g(a,b), c, g(c))$ is generated. de The figure*

[1] Indeed it is deterministic, but the difference is not relevant at the moment.

on the right of the automaton informally illustrates the relationship between the states,[2] *transitions and the data tree: a transition is denoted as a kind of tube. If some subtrees of the data tree have been recognized the automaton is in corresponding states. A transition is used, if the automaton is in all the corresponding input states (in the example below the tube) and the father node of the subtrees recognized with those states is labeled like the tube. The automaton is then not any longer in the states below the transition, but in the target state (precisely in all the target states of all the transitions traversable in that step). The root of the tree has to be accepted in such a way, that the resulting state is a final state. Note, that the two instances of the $\subset$ transition and the state q_2 denote the same objects in the automaton, they have been duplicated to illustrate acceptance of the input data that contains two subtree accepted by the same transition.*

$$
\begin{array}{lll}
A = & \{ & \{q_1,q_2,q_3,q_4,q_5,q_6,q_7,q_8\} \\
& , & \{f_{/3},a_{/0},b_{/0},c_{/0},g_{/2},g_{/1}\} \\
& , & \{q_4\} \\
& , & \{f(q_1(X),q_2(Y),q_3(Z)) \rightarrow q_4(f(X,Y,Z)) \\
& & ,g(q_5(X),q_6(Y)) \rightarrow q_1(g(X,Y) \\
& & ,c \rightarrow q_2(c) \\
& & ,a \rightarrow q_5(a) \\
& & ,b \rightarrow q_6(b) \\
& & ,g(q_2(X)) \rightarrow q_3(g(X)) \ \} \\
& \} &
\end{array}
$$



Acceptance Procedure The acceptance procedure recognizes, if a given tree is member of the tree language represented by a given automaton. A tree t is in the language $\mathscr{L}(A)$, if it is accepted by A. The acceptance procedure can be defined as non-deterministic algorithm expressed by a set of rules. The rules relate so called configurations of an automaton to each other. A configuration is a tree on which some nodes are annotated with a state, more formally $c \in T(\Sigma \cup Q)$ – note, that Q is defined as unary states, eg. a state can be seen as a (special) node in a tree with exactly one child node.

The rules have the following general shape:

$$
\frac{\begin{array}{c} C_1 \\ \vdots \\ C_n \\ t \end{array}}{t'} \qquad \text{(EXAMPLERULE)}
$$

where t and t' denote configurations. C_i denotes constraints on the configurations, part of them or their sub trees. The style of rules presented here is inspired by Gentzen or tableaux calculus rules and is often used in the context of type system formalization [4]. Whenever t is matched in the current configuration, t can be replaced by t'. The rules are

[2] It corresponds loosely to what will later on be introduced as *"aggregated acceptance path in the derivation tree"*.

applied until no rule is applicable anymore resulting in a sequence of configurations. If it is possible that more than one rules is applicable on one configuration (which is usually the case), a tree of possible configurations exists with sequences of configurations as paths through the tree.

The use of rules to express the acceptance procedure with finite automata is not common, yet useful to introduce the rule formalism, that will be used throughout this thesis in different places.

Rules for Acceptance Procedure based onthe Finite Tree Automata A given tree t is member of a language $\mathscr{L}(A)$ for an automaton $A = (Q, \Sigma, Q_F, \Delta)$, if there is a derivation of configurations based on the following rules with at least one closed branch of the derivation tree.

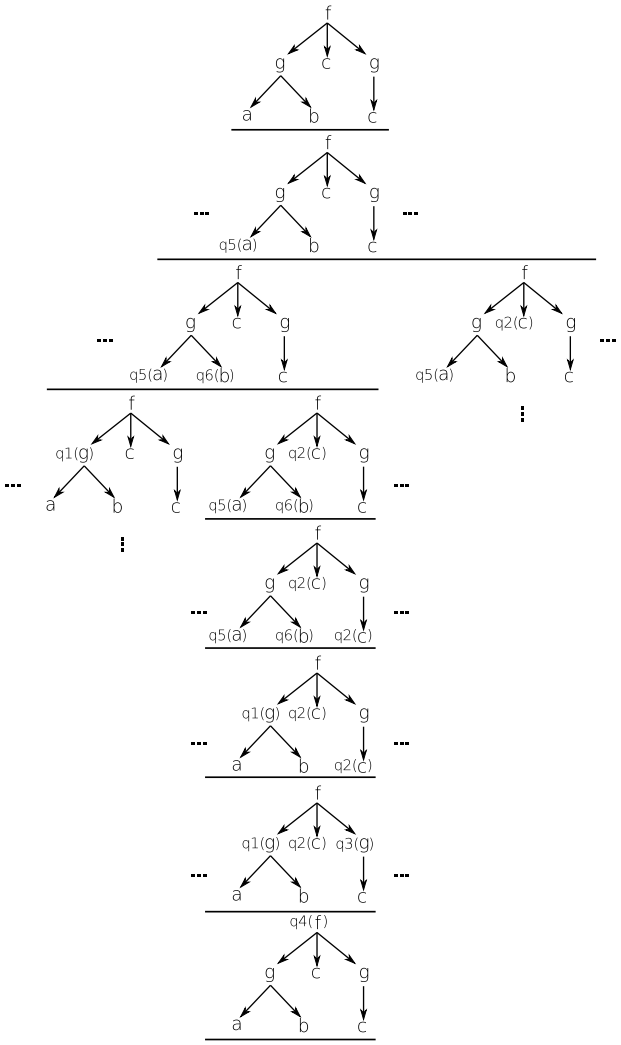
$$\frac{\begin{array}{c} q \in Q_F \\ t \in T(\Sigma) \\ q(t) \end{array}}{} \quad \text{(ROOT)}$$

The first rule states, that the given tree t is accepted, when a configuration is derivable such that t is accepted with a final state $q \in Q_F$. A branch of configuration derivations is succesfully closed. At least one successfully closed branch is necessary to prove membership of a givrn tree in the language represented by the given automaton.

$$\frac{\begin{array}{c} u_i \in T(\Sigma) \\ f \in \Sigma_n \\ f(q_1(X_1), \ldots, q_n(X_n)) \rightarrow q(f(X_1, \ldots, X_n)) \ \in \ \Delta \\ q, q_1, \ldots, q_n \in Q \\ f(q_1(u_1), \ldots, q_n(u_n)) \end{array}}{q(f(u_1, \ldots, u_n))} \quad \text{(REC)}$$

The Rec rule relates two configurations, if the tree contains a (sub)tree matching the left hand side of a transition in Δ. The (sub)tree is then replaced by the subtree on the right side of the transition rule with all variables (eg. X_i) substituted with the bindings of the left hand side. The only difference in the two configurations is the annotation of nodes with states. This is due to the nature of the transitions – left and right hand side are identical except of the change of intermediate state labeled branch parts.

Example 2 *Given the tree $f(g(a,b),c,g(c))$ and the example automaton A presented above, the following derivation justifies the recognition of the tree as an instance of the language represented by A:*



Aggregated Acceptance Path of the Derivation Tree Given a path of rule applications that proves the membership of a tree in the language of the corresponding automaton, the aggregated acceptance path is the tree resulting, when aggregating all the configurations of the path to one configuration such that all state annotations interlaced with the path are part of this configuration. This *artificial* configuration gives the information, which node was accepted with which transition. The transition can then be seen as some sort of type annotation for the nodes of the tree. Later on (see **??**), for type checking Xcerpt, this is used to deduce the types of nodes, as the transitions are shown to be related to grammar rules and therefore to grammar non terminal symbols which in turn represent types or type names.

ε-rules It is possible to extend the non-deterministic regular tree automata with ε-rules. Those are rules of the form $q \rightarrow q'$. Yet ε-rules are convenient in some cases (eg. for construction of an automaton based on some regular expression like formalism as shown in **??**), they do not enhance or restrict the expressiveness of non-deterministic

regular tree automata.[3]

deterministic finite tree automata Another common variant of non-deterministic finite tree automata are *deterministic finite tree automata*. A tree automaton $A = (Q, \Sigma, Q_f, \Delta)$ is deterministic (DFTA) if there are no two rules with the same left-hand side (and no ε-rules). Many text book approaches of standard operations on automata like intersection and union require deterministic automata. It is always possible to get a deterministic automaton of a non deterministic one, yet the resulting automaton may be of exponential size with respect to the input.

1.1 Handling Ranked Trees

For XML and any ordered semistructured data model, using regular tree automata for ranked trees is not possible without modificatio, as the data models are indeed unranked. A common way to handle unranked trees with tree automata is to map the unranked trees to ranked counterparts. A way to achieve this, is to lift tree nodes to a view, where nodes are represented eg. by a $node_{/3}$ item with the label as one child[4] of $node_{/3}$, the first child of the unranked tree as second child of $node_{/3}$ and the following sibling – if present – of the current node in the context of its parent node as third child of $node_{/3}$. As $node_{/3}$ is always of arity 3, it is necessary to provide an additional node type denoting the end of a branch, eg. the end of a list of siblings or an empty child list – this node will be called $eob_{/0}$ for *end of branch*.

Example 3 *The unranked tree $f[a, b[d], c]$ is mapped to the ranked counterpart*

$$node(f, node(a, eob, node(b, node(d, eob, eob), node(c, eob, eob))), eob)$$

A corresponding automaton has to recognize the ranked transcription of unranked trees the regular way.

Example 4 *An automaton accepting the former example could for example be:*

$$\begin{aligned}
A = \quad \{ \quad & \{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}\} \\
, \quad & \{node_{/3}, a_{/0}, b_{/0}, c_{/0}, eob_{/0}\} \\
, \quad & \{q_{11}\} \\
, \quad & \{ \ node(q_3(X), q_6(Y), q_6(Z)) \to q_7(node(X,Y,Z)) \\
& \ , \ a \to q_1(a) \\
& \ , \ b \to q_2(b) \\
& \ , \ c \to q_3(c) \\
& \ , \ d \to q_4(d) \\
& \ , \ f \to q_5(f) \\
& \ , \ eob \to q_6(eob) \\
& \ , \ node(q_4(X), q_6(Y), q_6(Z)) \to q_8(node(X,Y,Z)) \\
& \ , \ node(q_2(X), q_8(Y), q_7(Z)) \to q_9(node(X,Y,Z)) \\
& \ , \ node(q_1(X), q_6(Y), q_9(Z)) \to q_{10}(node(X,Y,Z)) \\
& \ , \ node(q_5(X), q_{10}(Y), q_6(Z)) \to q_{11}(node(X,Y,Z)) \ \} \\
\} &
\end{aligned}$$

[3] For an equivalence proof see [2], page 20.

[4] A label l is mapped to a node $l_{/0}$ in the ranked mapping.

2 An Automaton Model for Unranked Regular Rooted Graph Languages

In this section an automaton model for R_2G_2 is the inroduced. As R_2G_2 models languages of unranked trees, handling of unranked ordered trees is essential for the automaton model sought of.

As the class of tree grammars in use can be captured solely using automata operating on unranked tree transcriptions of ranked ones, it is useful to introduce an automaton model solely coping with such kind of languages. A new hypergraph based formalism is introduced. This formalism has proven useful for didactic purpose along this thesis as well as easy to implement. All methods involving data handling (eg. validation or typing) with automata are formulated directly on the unranked data formalism. For this reason, the automata are considered to be automata for unranked tree, opposed to automata for ranked trees as presented in [2].

2.1 Labelled Directed Hypergraphs as Non-Deterministic Regular Tree Automata

A non-deterministic regular tree automaton M is a 5-tuple (Q,Δ,F,R,Σ) with label alphabet Σ, states Q, final states F where $F \subseteq Q$, transitions Δ where $\Delta \subseteq (Q\times\Sigma\times Q\times Q) \cup (Q\times Q)$ (regular transitions are of the domain $Q\times\Sigma\times Q\times Q$ and ε-transitions are of the domain $Q\times Q$) and a set of root transitions R whith $R \subseteq \Delta$.[5]

For an automaton $A=(Q,\Delta,F,R,\Sigma)$ projection of the components is defined as $Q_A = Q$, $\Delta_A = \Delta$, $F_A = F$, $R_A = R$ and $\Sigma_A = \Sigma$. The union of two automata A_1 and A_2 is defined as the pairwise union of its components, eg. $A_1 \cup A_2 = (Q_{A_1} \cup Q_{A_2}, \Delta_{A_1} \cup \Delta_{A_2}, F_{A_1} \cup F_{A_2}, R_{A_1} \cup R_{A_2}, \Sigma_{A_1} \cup \Sigma_{A_2})$. The difference of two automata is defined in a similar way, yet consistency of all transitions must be retained, eg. all states and symbols involved in transitions are defined in Q, respectively Σ. Useful functions for the construction of automata are the addition and subtraction of transitions to (or from) an automaton defined as follows: $A_1 + \tau = A_2$ such that $\tau = (s,l,c,e)$ and $A_1 \cup (\{s,c,e\},\{\tau\},\{\},\{\},\{l\}) = A_2$, $A_1 - \tau = A_2$ such that $\tau = (s,l,c,e)$ and $A_1 \setminus (\{s,c,e\},\{\tau\},\{\},\{\},\{l\}) = A_2$.

For the sake of concreteness an example automaton for the following grammar G is presented:

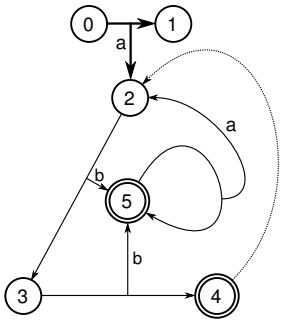
The language generated by the grammar

$$\begin{aligned} \texttt{element } A &\rightarrow a[\,(B,B)^+\,]; \\ \texttt{element } B &\rightarrow b[\,A^*\,]; \end{aligned}$$

is accepted by the following automaton

[5] Usually we need just one root transition, but for technical reasons it is convenient to have a set of root transitions.

$$
\begin{array}{lll}
A = (& \{ & 0,1,2,3,4,5\}, \\
 & \{ & (0,a,2,1), \\
 & & (2,b,5,3), \\
 & & (3,b,5,4), \\
 & & (4,2), \\
 & & (5,a,2,5)\}, \\
 & \{ & 4,5\}, \\
 & & (0,a,2,1), \\
 & \{ & a,b\})
\end{array}
$$



Comparing Hypergraph Automata and Ranked Tree Automata Applied to Unranked Data Transcription The hypergraph automaton model is a special notation for fixed arity tree automata where fixed arity nodes of arity 3 are matched – one position represents the label of the matched unranked node, one for the content list of this node and one for the following sibling node in the content list in which this node is contained. This is reflected in the hyperedges of arity 4 relating (1) the start state of the transition, (2) the label, (3) the start state of the child list and (4) the start state of the list of following siblings. The advantage of this approach is the arguably less bulky notation, as the node abstraction is not explicit. The disadvantage is the need to redefine many operations already available for ranked tree automata to hypergraph automata. From a practical point of view the hypergraphs are arguably well suited as automata models for type checking on Xcerpt.

Deterministic vs. Non-Deterministic Automata It is possible to restrict tree automata to *deterministic* tree automata – deterministic tree automata have always exactly one *matching* transition from a given left hand side to a new state along a given node label, while non-deterministic automata may have more than one matching transition. The decision procedure for membership test is simpler using deterministic automata, as all possible derivations (eg. paths of a decision tree) lead to a successfully closed branch. If no derivation rule is applicatbe any more, the data tree is not member of the language represented by the deterministic automaton. With non-deterministic automata, it is still possible, that earlier in the derivation tree another decision (eg. another choice of a transition) leads to a successfully closed branch. A deterministic algorithm checking membership using non-deterministic automata has therefore to retract choices in dead ends of the derivation tree, if further derivations are possible and membership has not been proved at that moment. The property of not having to retract derivation choices is called *confluence*. The decision procedure for membership test on deterministic automata is a confluent system.

While deterministic automata are favorable for membership testing, their creation from regular expressions may be of exponential complexity. Generation of non-deterministic finite automata based on regular expression as language specification can be done in polynomial time.[6] As R_2G_2 uses regular expressions to specify content models, the translation of R_2G_2 to deterministic finite automata may be of exponential complexity.

Fortunately, all necessary operations for type checking, eg. intersection, emptyness test, subset test, can be implemented in polynomial complexity directly on non-deterministic automata. This will be shown along this chapter when introduced.

[6] With respect to the size of the regular expression.

2.2 Membership Test for a Tree using Hypergraph Automata

An algorithm able to test membership of unranked trees in a language represented by a hypergraph based automaton is presented. In contrast to the standard approach for ranked tree as shown in [2] and introduced earlier, this algorithm is able to operate directly on the unranked tree model without prior transcription of data instances to a ranked tree representation. Calculus rules are used to explain the algorithm in a non-deterministic way. Rules are of the following shape:

$$\frac{\begin{array}{c} C_1 \\ \vdots \\ C_n \\ e_1 : a_1 \qquad e_n : a_n \end{array}}{e : a} \tag{EXAMPLE}$$

C_i denote constraints on e, a, e_i, a_i and e, e_i are trees or content lists of trees, ie. sequences of trees that all share the same parent node. By a, a_i either states or transitions of an automaton are denoted. An expression $e : a$ will also be called a configuration of the automaton. The rules relate configurations of automata. Two different kinds of configuration exist: (1) configurations of shape $t : \tau$ where t is a tree and τ is a transition, (2) or $[t_1, \ldots, t_n] : S$ where $[t_1, \ldots, t_n]$ is a list of trees and S is a state of the automaton.

$$\frac{\begin{array}{c} \tau \in R_A \\ (s, l, c, e) = \tau \\ se \in F_A \\ t : \tau \end{array}}{\qquad} \tag{ROOT}$$

The ROOT rule matches the root of the data tree, if there is a transition in the set of root transitions from which on a whole derivation tree can be found.

$$\frac{c \in F_A}{[\,] : c} \tag{END}$$

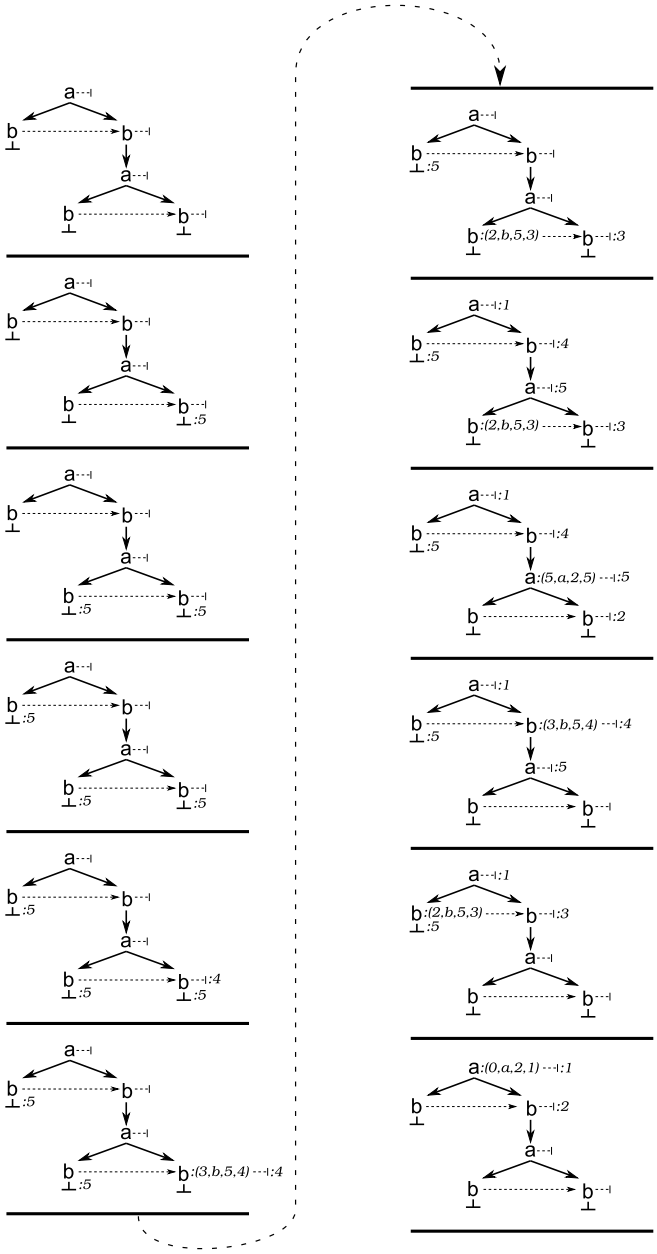
The END rule accepts an empty list, if the configuration involves an empty list and a state of the set of final states.

$$\frac{\begin{array}{c} \tau \in \Delta_A \\ \tau = (s, l, c, e) \\ [t_1, \ldots, t_n] : c \end{array}}{l[t_1, \ldots, t_n] : \tau} \tag{NODE}$$

$$\frac{\begin{array}{c} n \geq 1 \\ \tau \in \Delta_A \\ \tau = (s, l, c, e) \\ t_1 : \tau \qquad [t_2, \cdots, t_n] : e \end{array}}{[t_1, t_2, \ldots, t_n] : s} \tag{LIST}$$

In a successful derivation, applications of the NODE and the LIST rule are interwoven and all branches end with an application of the END rule while the root of the derivation tree is an application of the ROOT rule. A tree without possible derivation is not valid with respec to the given automaton, multiple derivations may exist.

Example of a Tree Recognition using a Hypergraph Automaton Given the tree $a[b[],b[a[b[],b[]]]]$ and the automaton A as presented in the former example (example **??**, the following derivation is a possible recognition:



Operational Semantics of The Recognition Rules The rules presented above give an abstract description of a recognition algorithm. Neither the control flow nor decision in case of ambiguity are captured by the rules. It is possible, that an automaton can recognize a data tree using different derivations, a concrete algorithm should be designed

to either choose one of those derivations, maybe driven by other parameters.

Exemplary, a simple algorithm choosing one rule is sketched now:

Algorithm 2.1: MEMBERSHIPTEST($A = (Q, \Sigma, \Delta, S, F), e : a, \rho$)

comment: ρ denotes the set of rules

comment: e denotes either a tree or a list of trees

comment: either $a \in Q$ or $a \in \Delta$

comment: To check a tree t, call MEMBERSHIPTEST(A,t:s,R) with $s \in S$

$$\begin{array}{l} C_1 \\ \vdots \\ C_m \end{array}$$

for $\dfrac{e_1 : a_1 \quad e_n : a_n}{e_\rho : a_\rho} \in R$

do
- **if** $C_1 \wedge \cdots \wedge C_m = true$
- **then**
  - **if** MEMBERSHIPTEST$(A, e_1 : a_1, \rho) = true$ $\wedge \cdots \wedge$ MEMBERSHIPTEST$(A, e_n : a_n, \rho) = true$
  - **then return** $(true)$

return $(false)$

2.3 Recognition of a Rooted Graph using Hypergraph Automata

The recognition of rooted graphs is defined in analogy to the recognition of trees – a rooted graph is recognized by an automaton, if it is in the language accepted by this automaton. There is a certain correlation of the acceptance of a word by an automaton with the recognition of equality of two words: for each word, it is easy to obtain an automaton such that exactly this word is accepted. Therefore, this automaton provides a way to decide about equality of two words. The most precise way to judge about equality of two graphs is graph isomorphism. The decision procedure for graph isomorphism has exponential complexity. Assuming, that we base a recognition procedure for graphs on graph isomorphism – a graph is accepted, if it is isomorph to a graph in the language accepted by the automaton – then the whle process has to have exponential complexity, as otherwise the graph isomorphism itself would have sub exponential complexity (eg. it could be reformulated by means of graph recognition)

A weaker kind of membership relation between graphs will be chosen: the simulation relation – a graph is accepted by an automaton, if there is an instance in the language accepted by the automaton that simulates the graph.

> A simulation preorder is a relation between [graphs] associating systems which behave in the same way in the sense that one system simulates the other. Intuitively, a system simulates another system if it can match all of its moves .[7]

[7] From `http://en.wikipedia.org/wiki/Simulation_preorder`.

A first advantage of using the simulation preorder as base of the membership test in the recognition procedure is, that the decision procedure can be achieved in polynomial time. Second advantage, Xcerpt is based on a non standard unification called *simulation unification*, which itself is based on simulation preorder. In [5] the simulation preorder on so called ground query terms – of which data trees and graphs are a subset – is presented including also complexity results. Ground Xcerpt query term simulation has been shown to be a useful relation between trees or graphs. Arguably simulation preorder reflects well a notion of *"expected result"* for many applications of querying Web and Semantic Web data.

(Possibly Infinite) Tree Representations of Graphs As the recognition procedure is defined for trees so far, it is arguably useful to base the handling of graph shaped data on tree recognition. Note, that trees are a special kind of graphs, so trivially those graphs are already handled. For directed, acyclic graphs it is always possible to find a spanning, finite tree, where nodes accessible from one node chosen as a root using different paths are duplicated in the tree representation. In general, a graph can be spanned by different trees, capturing different possible graph traversals. As the root in a rooted graph is fixed, there is only one possible such spanning tree. Cyclic graphs can conceptually be represented using infinite trees where infinite always means finite in breadth (a node in a finite graph can only have finite many successors, so can the corresponding node in the tree) and branches of infinite depth for cycles.

By applying the tree approach for recognition on acyclic graphs, an algorithm is achieved that possibly checks the same nodes multiple times, but that always terminates. It is possible, that the same node is checked multiple times using different or the same automama transitions. Arguably it is reasonable to *remember* acceptance results of nodes with corresponding transitions, as not only the testing of validity of a certain node in a context can be obmitted, but also the testing of all child nodes can be skipped. The process of *remembering* earlier calculations in this state is called memoisation.

By applying the tree approach to cyclic graphs, the recognition process gets stuck in non termination. However, the explained extension of memoisation guarantees termination. This is due to the fact, that any data tree node can in worst case only be tested against finite many transitions of the automaton, as the automata are finite.

An Algorithm for the Recognition of Rooted Graphs The former algorithm for tree recognition is now extended by memoisation to recognize rooted directed graphs. A graph is recognized by an automaton A, if it is simulated by a graph in the language $\mathscr{L}(A)$. The set of rules will not be affected by this change, but the algorithm for the application of the rules on a given data tree. This emphasizes the declarative nature of

the rules and the fact, that conceptually trees and graphs are handled in a similar way:

Algorithm 2.2: MEMBERSHIPTEST($A = (Q, \Sigma, \Delta, S, F), e : a, \rho$)

global *memo*
comment: ρ denotes the set of rules
comment: e denotes either a graph node or an adjecence list of graph nodes
comment: either $a \in Q$ or $a \in \Delta$
comment: To check a graph t call MEMBERSHIPTEST(A,t:s,R); $s \in S$ and $memo = \{\}$

if $e : a \mapsto b \in memo$
  then return (b)
  else
$memo \leftarrow memo \cup \{e : a \mapsto true\}$
C_1
$\vdots$
C_m
for $\dfrac{e_1 : a_1 \quad e_n : a_n}{e_\rho : a_\rho} \in \rho(2DO : is - it - \rho - or - R)$
do
if $C_1 \wedge \cdots \wedge C_m = true$
then
if MEMBERSHIPTEST$(A, e_1 : a_1, \rho) = true$
$\wedge \cdots \wedge$
MEMBERSHIPTEST$(A, e_n : a_n, \rho) = true$
then return $(true)$
$memo \leftarrow memo \setminus \{e : a \mapsto true\}$
$memo \leftarrow memo \cup \{e : a \mapsto false\}$
return $(false)$

2.4 Calculus for a Translation of R_2G_2 Definitions into Automata

Yet the presented automaton model is well suited as execution model for recognition of trees and graphs on regular rooted graph languages, it is not convenient as language definition formalism for the end user of applications of such problems – eg. for document schema authors and programmers. R_2G_2 arguably is an appropriate formalism for this task. An algorithm for translation of R_2G_2 language definitions into hypergraph based automata is presented now. The techniques used are a combination of two classical text book approaches: (1) content models – represented as regular expressions – are translated using the approach presented in [3], Chapter 3 [8] for translation of regular expressions into non-deterministic automata, (2) grammar rules are translated using a standard technique for the construction of NFA's as presented in [**?**], Chapter 1.[9]

The rules are shaped as follows:

[8] See the proof of theorem 3.7.
[9] See the proof of the theorem on page 35.

$$\frac{\begin{array}{c} C_1 \\ \vdots \\ C_n \end{array}}{A_1 \mid t_1 \mapsto (i_1,o_1) \quad \cdots \quad A_n \mid t_n \mapsto (i_n,o_n)} \qquad A \mid t \mapsto (i,o)$$ (EXAMPLERULE)

In addition, a global lookup function, called L, is used to map each type name (as found in the grammars), to a transition, therefore $L \subseteq N \times T$ where N is the set of type names in a R_2G_2 grammar as defined in **??**. The basic expressions related in the rules are of shape $A \mid t \mapsto (i,o)$. A, A_i denote automata. t, t_i denote terms or lists of terms. Usually the t's above the line in a calculus rule are decompositions of the t below the line. (i,o) is a tuple of states, they have to be part of the automaton A in the same context expression, ie. i and o have to occur in A in the context expression $A \mid t \mapsto (i,o)$. Additionally, the rules may contain constraints – denoted C_i in the example rule – over the components of the expressions encountering in a rule.

$$\frac{\begin{array}{c} L(N_j) = (s,l,c,e) \\ \rho_j = \texttt{element}\ N_j \rightarrow l[\cdots] \\ A_1 \mid \rho_1 \mapsto (i_1,o_1) \quad \cdots \quad A_n \mid \rho_n \mapsto (i_n,o_n) \end{array}}{(\{\},\{\},\{(s,l,c,e)\},\{e\},\{\}) \cup A_1 \cup \cdots \cup A_n \mid \rho_1,\cdots,\rho_n,\texttt{root} = N \mapsto (i_j,o_j)}$$ (GRAMMAR)

$$\frac{\begin{array}{c} L(N) = (s,l,s_{re},e) \\ A \mid re \mapsto (s_{re},e_{re}) \end{array}}{A \cup (\{s,e\},\{L(N)\},\{\},\{\},\{l\}) \mid \texttt{element}\ N \rightarrow l[\,re\,] \mapsto (s,e)}$$ (RULE)

$$\frac{A \mid re \mapsto (s_{re},e_{re})}{A \cup (\{s,e\},\{(s,l,s_{re},e)\},\{\},\{e_{re}\},\{l\}) \mid l[\,re\,] \mapsto (s,e)}$$ (TYPETERM)

$$\frac{L(N) = (_{s'},l,c,_{e'})}{A \cup (\{s,e\},\{(s,l,c,e)\},\{\},\{\},\{\}) \mid N \mapsto (s,e)}$$ (TYPENAME)

$$\frac{A_1 \mid re_1 \mapsto (s_{re_1},e_{re_1}) \qquad A_2 \mid re_2 \mapsto (s_{re_2},e_{re_2})}{A_1 \cup A_2 \cup (\{\},\{(e_{re_1},s_{re_2})\},\{\},\{\},\{\}) \mid re_1,re_2 \mapsto (s_{re_1},e_{re_2})}$$ (RESEQ)

$$\frac{\begin{array}{c} A = (\{s,e\},\{(s,s_{re_1}),(s,s_{re_2}),(e_{re_1},e),(e_{re_2},e)\},\{\},\{\},\{\}) \\ A_1 \mid re_1 \mapsto (s_{re_1},e_{re_1}) \qquad A_2 \mid re_2 \mapsto (s_{re_2},e_{re_2}) \end{array}}{A_1 \cup A_2 \cup A \mid re_1 | re_2 \mapsto (s,e)}$$ (REDISJ)

$$\frac{A \mid re \mapsto (s_{re},e_{re})}{A \cup (\{\},\{(s_{re},e_{re})\},\{\},\{\},\{\}) \mid re^? \mapsto (s_{re},e_{re})}$$ (REOPT)

$$\frac{A \mid re \mapsto (s_{re},e_{re})}{A \cup (\{\},\{(e_{re},s_{re})\},\{\},\{\},\{\}) \mid re^+ \mapsto (s_{re},e_{re})}$$ (REPLUS)

$$\frac{A \mid re \mapsto (s_{re},e_{re})}{A \cup (\{\},\{(s_{re},e_{re}),(e_{re},s_{re})\},\{\},\{\},\{\}) \mid re^* \mapsto (s_{re},e_{re})}$$ (REKLEENE)

TODO:
? possibly the rules need more explanation...

2.5 The Emptyness Test

The emptyness test finds out, if for an automaton, there may be any data instance accepted by this automaton, if therefore the language accepted by the automaton is non empty. For an automaton to accept finite trees, it is obviously neccesary to find paths along the hyper edges ending in final states. For infinite trees or graphs containing loops, this property can be relaxed, as such data instances can be accepted by loops without final state in the automaton. Automata constructed from R_2G_2 definitions arguably always accept non empty languages for three reasons: (1) As they have a root transition by definition (based on the mandatory `root` declaration). (2) Along the breadth axis of the automata there is always either an end state at the end of each path or the path is a loop containing an end state. This is due to the fact, that the last state constructed by regular expression decomposition is always an end state, or a looping ε-edge is added to an end state terminated path to express repetition. (3) Along the depth axis there is either an end state due to empty content, or a transition to a state representing another grammar rule. This state again is part of a non empty breadth axis and either of a final state terminated depth axis or of a depth axis recursively fulfilling reason 3. A depth axis loop without final state can therefore only accept infinite trees or graphs containing an appropriate loop.

Nevertheless, automata representing only empty languages exist in practise: intersection of two automata can lead to an automaton accepting only empty languages, eg. by construction of an automaton without start transition or by construction of an automaton with root transitions with all outgoing edges ending only in branches without loops and final states. Detection of automata representing empty languages is important for type checking.

An algorithm for detection of emptyness for a given automaton is sketched now:

Algorithm 2.3: IsEmpty($A = (Q, \Sigma, \Delta, S, F)$)

memoisation ←create a lookup table of truth values with index over Q
comment: *memoisation* is defined in each call of IsEmpty().

procedure RecursiveTransitionTest($\delta = (s, l, c, e)$)
return (DepthTest(c) ∧ BreadthTest(e))

procedure RecursiveTransitionTest($\delta = (s, e)$)
return (BreadthTest(e))**comment:** handling of ε-transitions.

procedure BreadthTest(v)
if $v \in$ *memoisation*
then return (*memoisation*[v])
memoisation[v] ← *true*
for $(v, l, c, e) \in \Delta$
do { **if** RecursiveTransitionTest((v, l, c, e)) = *false*
then { *memoisation*[v] ← *false*
return (*false*)
exit
return (*true*)

procedure DepthTest(v)
if $v \in$ *memoisation*
then return (*memoisation*[v])
memoisation[v] ← *false*
for $(v, l, c, e) \in \Delta$
do { **if** RecursiveTransitionTest((v, l, c, e)) = *false*
then { *memoisation*[v] ← *false*
return (*false*)
exit
memoisation[v] ← *true*
return (*true*)

main
for $\delta \in S$
do { **if** RecursiveTransitionTest(δ) = *false*
then { **return** (*false*)
exit
return (*true*)

2.6 Intersection of Regular Rooted Graph Automata

Calculating the intersection of two regular languages is a common exercise in text books about theoretical computer science and automata theory and it is also of high practical use. Given eg. two language definitions for two versions of a data format, the

intersection reflects a kind of conservative transitional data format providing guaranteed backward and forward compatibility. In type checking of the Xcerpt query language, non empty intersection can play an important role for checking selection constructs: given a query with multiple occurences of the same variable, the occurences may have different types. If the types have empty intersection, no data exists conforming the type constraint of the variables, therefore the selection may never select any valid data with respect to the types and is therefore arguably useless. Note, that different type annotations may either occur due to a query programmers annotation or due to type inference. Checking consistency of such concurrent type annotations in Xcerpt query terms can also be handled using an emptyness test for the type intersection.

Intersection of Regular (String) Languages Using DFAs The presented approach is a classical text book approach as found in [3]. It serves as introduction to a technique of calculating intersection and will be modified to non-deterministic and then to regular graph automata.

It is easily possible to construct the intersection of L_1 and L_2, if union and complement are defined, as generally $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ holds. A direct construction is presented, as neither union, nor complement is presented by now, and is not strictly necessary for type checking of Xcerpt later on. A direct construction is achieved by simulating parallel execution of the two deterministic finite automata representing L_1 and L_2. This corresponds to the construction of the product automaton:

Let deterministic automata be defined as 5-tuples $(Q, \Sigma, \Delta, s, F)$ with Q as the states of the automaton, Σ as the alphabet of the corresponding language, s the start state and $F \subseteq Q$ as the final states. The transitions $\Delta \subset Q \times \Sigma \times Q$ are defined such that for every $v \in Q$ and for every $l \in \Sigma$ there is a transition $(v, l, v') \in \Delta$ and no other transition $(v, l, v'') \in \Delta$ with $v', v'' \in Q$ and $v' \neq v''$.

For L_1 accepted by $A_1 = (Q_1, \Sigma_1, \Delta_1, s_1, F_1)$ and L_2 accepted by $A_2 = (Q_2, \Sigma_2, \Delta_2, s_2, F_2)$, the intersection $L_1 \cap L_2$ is accepted by $A_\cap = (Q_1 \times Q_2\ ,\ \Sigma_1 \cap \Sigma_2\ ,\ \Delta_\cap\ ,\ (s_1, s_2)\ ,\ F_1 \times F_2)$ where $\Delta_\cap((p, p'), a, (q, q')) = (\Delta_1(p, a, q), \Delta_2(p', a, q'))$.

See Figure 1 for an example on how to get a product of two automata.

An algorithm for the construction of an automaton $A_\cap = (Q_\cap, \Sigma_\cap, \Delta_\cap, s_\cap, F_\cap)$ from two automata $A_1 = (Q_1, \Sigma_1, \Delta_1, s_1, F_1)$ and $A_2 = (Q_2, \Sigma_2, \Delta_2, s_2, F_2)$ is presented now:

Algorithm 2.4: INTERSECTIONDFA(A_1, A_2)

$\Sigma_\cap \leftarrow \Sigma_1 \cap \Sigma_2$
$s_\cap \leftarrow (s_1, s_2)$
$Q_\cap \leftarrow Q_1 \times Q_2$
$F_\cap \leftarrow \{(v_1, v_2) \in Q_\cap \mid v_1 \in F_1 \wedge v_2 \in F_2\}$
for $(v_1, l, v_1') \in \Delta_1$
do { **for** $(v_2, l, v_2') \in \Delta_2$
do $\{\Delta_\cap \leftarrow ((v_1, v_2), l, (v_1', v_2'))$

Extending the Approach to Non-Deterministic Finite Automata The presented approach has the drawback to require deterministic automata, that may have exponential size of a corresponding non-deterministic automaton. The automaton model focused on in this thesis are usually non-deterministic ones. Fortunately, the approach can be extended to non-deterministic automata without exponential blowup in time or space.

Figure 1: The product automaton on the right accepts the intersection of the language of the two automata on the left.

The difference between deterministic and non-deterministic automata in a nutshell is (1) non-deterministic automata may have spontaneous state transitions along so called ε-edges without consumption of an input symbol, and (2) while each symbol in Σ has exactly one outgoing transition from each state in deterministic automata, any number of such edges may occur in the non-deterministic case.

Let non-deterministic automata be defined as 5-tuples $(Q, \Sigma, \Delta, s, F)$ with Q as the states of the automaton, Σ as the alphabet of the corresponding language, s the start state and $F \subseteq Q$ as the final states. The transitions are defined as $\Delta \subseteq ((Q \times \Sigma \times Q) \cup \Delta_\varepsilon)$ with $\Delta_\varepsilon \subseteq (Q \times Q)$.

To simulate the parallel execution of two automata in a product automaton with an epsilon transition (a, e) in one automaton, it is necessary to provide an epsilon edge for any product state (a, v) to the corresponding state (e, v). This reflects the possibility of a spontaneous transition every time the automaton with $a \in Q$ is in state a, independent of the state of the other automaton.

To handle the arbitrary amount of edges with one label from a state, no further change is necessary, as the deterministic algorithm already relates *all* edges of one automaton with all edges of the other one, as long as the transition labels match. In the deterministic case, by definition only one edge per state and label exists, therefore the same algorithm behaves as defined for the deterministic case.

An algorithm for the construction of an automaton $A_\cap = (Q_\cap, \Sigma_\cap, \Delta_\cap, s_\cap, F_\cap)$ from two non-deterministic automata $A_1 = (Q_1, \Sigma_1, \Delta_1, s_1, F_1)$ and $A_2 = (Q_2, \Sigma_2, \Delta_2, s_2, F_2)$ is

presented now:

Algorithm 2.5: INTERSECTIONNFA(A_1,A_2)

$\Sigma_\cap \leftarrow \Sigma_1 \cap \Sigma_2$
$s_\cap \leftarrow (s_1, s_2)$
$Q_\cap \leftarrow Q_1 \times Q_2$
$F_\cap \leftarrow \{(v_1, v_2) \in Q_\cap \mid v_1 \in F_1 \wedge v_2 \in F_2\}$
for $(v_1, l, v_1') \in \Delta_1$
  do { **for** $(v_2, l, v_2') \in \Delta_2$
    do $\Delta_\cap \leftarrow \Delta_\cap \{((v_1, v_2), l, (v_1', v_2'))\}$
for $(v_1, v_1') \in \Delta_1$
  do { **for** $v_2 \in Q_2$
    do $\Delta_\cap \leftarrow \Delta_\cap \cup \{((v_1, v_2), (v_1', v_2))\}$
for $(v_2, v_2') \in \Delta_1$
  do { **for** $v_1 \in Q_1$
    do $\Delta_\cap \leftarrow \Delta_\cap \cup \{((v_1, v_2), (v_1, v_2))\}$

Extending the Approach to Graph Automata The main difference of string- and graph automata is the shape of the transitions – triples for string automata and quatruples for graph automata. Fortunately, the calculation of an automaton accepting the language intersection of two automata, is easily derivable from the string automaton case. Informally, the only difference is the handling of the third state.

Algorithm 2.6: INTERSECTIONNDFTA(A_1,A_2)

$\Sigma_\cap \leftarrow \Sigma_1 \cap \Sigma_2$
let $(a_1, l_1, c_1, e_1) = s_1$
let $(a_2, l_2, c_2, e_2) = s_2$
$s_\cap \leftarrow ((a_1, a_2), l, (c_1, c_2), (e_1, e_2))$
$Q_\cap \leftarrow Q_1 \times Q_2$
$F_\cap \leftarrow \{(v_1, v_2) \in Q_\cap \mid v_1 \in F_1 \wedge v_2 \in F_2\}$
for $(v_{a_1}, l, v_{c_1}, v_{e_1}) \in \Delta_1$
  do { **for** $(v_{a_2}, l, v_{c_2}, v_{e_2}) \in \Delta_2$
    do $\Delta_\cap \leftarrow \Delta_\cap \{((v_{a_1}, v_{a_2}), l, (v_{c_1}, v_{c_2}), (v_{e_1}, v_{e_2}))\}$
for $(v_1, v_1') \in \Delta_1$
  do { **for** $v_2 \in Q_2$
    do $\Delta_\cap \leftarrow \Delta_\cap \cup \{((v_1, v_2), (v_1', v_2))\}$
for $(v_2, v_2') \in \Delta_1$
  do { **for** $v_1 \in Q_1$
    do $\Delta_\cap \leftarrow \Delta_\cap \cup \{((v_1, v_2), (v_1, v_2))\}$

Figure 2 illustrates the cross product of two automata. The automata accept the languages defined by the grammar (for the upper left automaton)

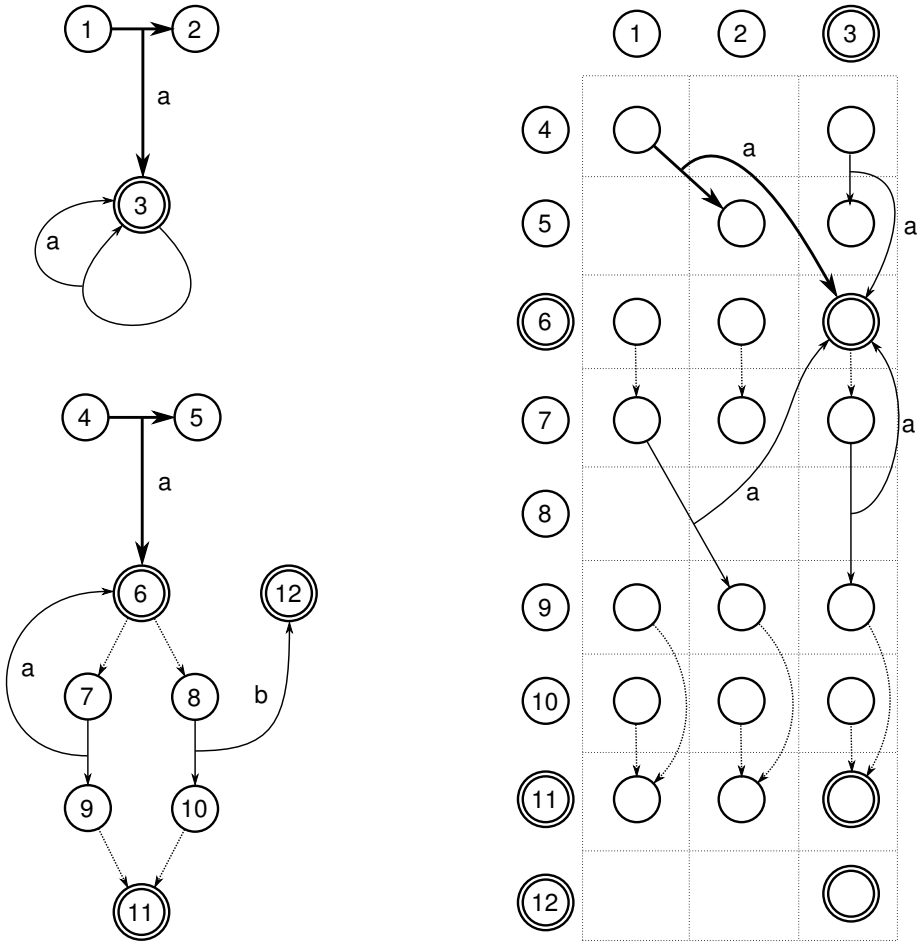


Figure 2: The product automaton on the right accepts the intersection of the graph language of the two automata on the left.

```
root A;
element A = a[ A* ];
```

and the grammar (for the lower left automaton)

```
root A;
element A = a[ A|B ];
element B = b[ ];
```

The resulting automaton accepts the language represented for example by the grammar

```
root A;
element A = a[ A ];
```

The resulting intersection automaton contains some unreachable states and transitions, that could easily be removed using some minimization algorithm or simply by

applying a reachability algorithm. As this is not essential to the tractability of type checking later on, automata minimization will not be considered.

2.7 Automata Based Subset Test for two Regular Rooted Graph Languages

Given a regular language, testing if it is a subset of another regular language, is an important task e.g. in type checking. If eg. it is possible to infer the type of a variable used in the output or construction part of a query (maybe the type is implied by a selection), this variable is well typed with respect to a given type, if the inferred type is a subtype of the type given by the programmer.

Another very practical use case is schema checking for special document schemata: if one wants to make a schema for HTML documents of a certain shape, e.g. a webpage supporting the corporates look and feel by using certain navigation elements, testing that this schema represents a subset of HTML is desirable.

The approach for subset testing presented here is based on simulation preorder as defined in [5]. A simulation preorder is a relation between state transition systems associating systems which behave in the same way in the sense that one system simulates the other. Formally, given a state transition system with states S, a simulation preorder is a binary relation $R \subseteq S \times S$ such that if $(p,q) \in R$, then for each transition $p \xrightarrow{a} p'$ there is a transition $q \xrightarrow{a} q'$ such that $(p',q') \in R$.

For string language automata (DFA's or NFA's) A_1 and A_2, simulation preorder is specialised in such a way, that A_1 and A_2 are in simulation preorder – written $A_1 \preceq A_2$ later on – if each initial state of A_1 simulates in an initial state of A_2 and for each final state of A_1 there is a final state in A_2 in which it simulates. For automata defined as $A = (S,T,F,s_0,\Sigma)$ where S is the set of states, $T \subseteq S \times \Sigma \times S$ is the set of transitions, $s_0 \in S$ is the start state and $F \subseteq S$ is the set of final states, the definition of the simulation preorder over label equality can be written as:

$$\begin{array}{rcl} A_1 \preceq A_2 & \Rightarrow & \forall (s,l,e) \in T_{A_1} \exists (s',l',e') \in T_{A_2}.(s,l,e) \preceq (s',l',e') \\ (s,l,e) \preceq (s',l',e') & \Rightarrow & l = l' \ \wedge \\ & & \forall (e,\bar{l},\bar{e}) \in T_{A_1} \exists (e',\bar{l}',\bar{e}') \in T_{A_2}.(e,\bar{l},\bar{e}) \preceq (e',\bar{l}',\bar{e}') \end{array}$$

Extending the definition of automata simulation to graph automata simulation is strait forward: the recursive $\preceq$ condition is tested along both dimensions of the hyper edges as used in the tree automata:

$$\begin{array}{rcl} A_1 \preceq A_2 & \Rightarrow & \forall (s,l,c,e) \in T_{A_1} \exists (s',l',c',e') \in T_{A_2}.(s,l,c,e) \preceq (s',l',c',e') \\ (s,l,c,e) \preceq (s',l',c',e') & \Rightarrow & l = l' \ \wedge \\ & & \forall (e,\bar{l},\bar{c},\bar{e}) \in T_{A_1} \exists (e',\bar{l}',\bar{c}',\bar{e}') \in T_{A_2}.(c,\bar{l},\bar{c},\bar{e}) \preceq (c',\bar{l}',\bar{c}',\bar{e}') \ \wedge \\ & & \forall (e,\vec{l},\vec{c},\vec{e}) \in T_{A_1} \exists (e',\vec{l}',\vec{c}',\vec{e}') \in T_{A_2}.(e,\vec{l},\vec{c},\vec{e}) \preceq (e',\vec{l}',\vec{c}',\vec{e}') \end{array}$$

2.7.1 An Algorithm for Subset Graph on Tree Automata

As a sketch for implementation and for complexity analysis of the presented simulation relation on tree automata, the following algorithm is proposed. The algorithm is applied to two automata A_1 and A_2:

- a two dimensional matrix of truth values of size $|T_{A_1}| \times |T_{A_1}|$ is initialized in such a way, that for each transition pair $(\tau_1,\tau_2) \in T_{A_1} \times T_{A_1}$ the corresponding field in the matrix is $true$, if the labels of τ_1 and τ_2 are identical and $false$ otherwise.

- set each matrix field with value $true$ to $false$, if $((s_1,l,c_1,e_1),(s_2,l,c_2,e_2))$ is the corresponding transition pair and either s_1, c_1 or e_1 is a final state but not the corresponding s_2, c_2 or e_2.
- modify the matrix until a fixpoint is reached by
  - set each matrix field with value $true$ in the matrix with corresponding transition pair $((s_1,l,c_1,e_1),(s_2,l,c_2,e_2))$ to $false$, if for any transition $\tau_1 = (e_1, \vec{l}, \vec{c_1}, \vec{e_1})$ in A_1 there is no corresponding transition $\tau_2 = (e_2, \vec{l}, \vec{c_2}, \vec{e_2})$ in A_2 such that the field (τ_1,τ_2) in the matrix is $true$.
  - set each $true$ field in the matrix with corresponding transition pair $((s_1,l,c_1,e_1),(s_2,l,c_2,e_2))$ to $false$, if for any transition $\tau_1 = (c_1,\bar{l},\overline{c_1},\overline{e_1})$ in A_1 there is no corresponding transition $\tau_2 = (c_2,\bar{l},\overline{c_2},\overline{e_2})$ in A_2 such that the field (τ_1,τ_2) in the matrix is $true$.
- If for any transition $\tau_1 \in A_1$ there is no correspondig transition $\tau_2 \in A_2$ such that (τ_1,τ_2) in the matrix is $true$, then the language accepted by A_1 is not a subset of the language accepted by A_2.

Example 5 *The condition $A_1 \subseteq A_2$ is to be tested using the presented algorithm.*

$$
\begin{aligned}
A_2 = (\ & \{\ 1,2,3,4,5,6,7,8,9\},\\
& \{\ (1,a,3,\boldsymbol{2}),\\
& \quad (3,a,3,\boldsymbol{4}),\\
& \quad (4,a,3,\boldsymbol{6}),\\
& \quad (\boldsymbol{6},b,\boldsymbol{6},7),\\
& \quad (7,c,\boldsymbol{8},\boldsymbol{9}),\\
& \quad (6,b,5,\boldsymbol{6}),\\
& \quad (5,a,3,\boldsymbol{6}),\\
& \quad (3,a,3,\boldsymbol{6})\},\\
& \{\ 2,6,8,9\},\\
& \quad (1,a,3,\boldsymbol{2}),\\
& \{\ a,b,c\})
\end{aligned}
\qquad
\begin{aligned}
A_1 = (\ & \{\ 10,11,12,13,14,15\},\\
& \{\ (10,a,12,\boldsymbol{11}),\\
& \quad (12,a,12,\boldsymbol{14}),\\
& \quad (\boldsymbol{14},b,15,\boldsymbol{14}),\\
& \quad (\boldsymbol{13},a,12,15)\},\\
& \{\ 11,13,14\},\\
& \quad (10,a,12,\boldsymbol{11}),\\
& \{\ a,b\})
\end{aligned}
$$

A run of the algorithm is visualized with a table representing the matrix. The edges of A_1 are used as column labels and the edges of A_2 as row labels. Final states are emphasized using a ***bold*** *font. The cells contain a series of ones (1) and zeros (0) representing the truth values $true$ and $false$ a field has in various stages of the computation. Note, that if a 0 occurs in the cell, the 0 is the ultimate value of this cell, as the algorithm only changes $true$ values, in case of conflicts, to $false$ values. 4 states of computations are represented, so either a cell contains $1,1,1,1$ and is thereby $true$, or it contains less enties where the last state is 0, eg. $1,1,0$. The stages represented are:*

1. *after performing the label check,*
2. *after checking, that final states in transitions of A_1 fall on final states of corresponding transitions of A_2*
3. *first iteration of checking following transitions in both dimensions (two cells changed truth value)*
4. *second (and last) iteration of checking following transitions in both dimensions (no cells changed truth value)*

$\downarrow A_2 \quad A_1 \rightarrow$	$(10,a,12,\mathbf{11})$	$(12,a,12,\mathbf{14})$	$(\mathbf{14},b,15,\mathbf{14})$	$(15,a,12,\mathbf{13})$
$(1,a,3,\mathbf{2})$	$1,1,1,1$	$1,1,0$	0	$1,1,1,1$
$(3,a,3,4)$	$1,0$	$1,0$	0	$1,0$
$(4,a,3,\mathbf{6})$	$1,1,1,1$	$1,1,1,1$	0	$1,1,1,1$
$(\mathbf{6},b,7,\mathbf{6})$	0	0	$1,1,0$	0
$(7,c,\mathbf{8},\mathbf{9})$	0	0	0	0
$(\mathbf{6},b,5,\mathbf{6})$	0	0	$1,1,1,1$	0
$(5,a,3,\mathbf{6})$	$1,1,1,1$	$1,1,1,1$	0	$1,1,1,1$
$(3,a,3,\mathbf{6})$	$1,1,1,1$	$1,1,1,1$	0	$1,1,1,1$

After the last iteration, the columns are checked for consistency, ie. each column should contain at least one true cell, so a table cell with $1,1,1,1$. *As this is the case,* A_1 *is an automaton accepting a sublanguage of the language represented by* A_2.

An Upper Bound Complexity for Subset Test The presented algorithm shows, that the subset test has an upper bound of polynomial time and space complexity. The space complexity is determined by the matrix that is of the size of the product of the number of transitions of both automata, ie. $O(|\Delta_1| \times |\Delta_2|)$. The time complexity is the sum of initializing the matrix (including label test and final state condition) and the iterative refinement of the matrix. The refinement process must terminate, as either no change is made to the matrix and then the refinement is over, or at least one cell changes truth value from $true$ to $false$. Truth values are never altered from $false$ to $true$ again. Assuming the worst case, that on each iteration process just one cell is altered, we need $|\Delta_1| \times |\Delta_2|$ iterations, each iteration has a complexity of $O(|\Delta_1| \times |\Delta_2|)$. The final step is the consistency check of the columns, which also takes $O(|\Delta_1| \times |\Delta_2|)$ time. The total costs therefore are $O(\ (k + (|\Delta_1| \times |\Delta_2|)) \times (|\Delta_1| \times |\Delta_2|)\ )$ with k as factor for initialisation cost and consistency check of one cell.

3 Current Work

Currently the complexity of the various algorithms presented here is being analysed. Further, research on how to represent unordered content models and on how to reason on them is going on. Constraint solving techniques on diophantic solutions of linear equation systems will be used for this purpose.

Some hypothesis about the complexity of some of the forlerly presented algorithms is presented below.

3.1 An Upper Bound Complexity for Membership Test of Tree Shaped Data

Let N be the number of nodes in the tree and M the number of types. Each node may have at most 1 type, as the algorithm stops on success. On each node we may have to check-and-fail M types, and on each subnode as well. If subnode membership failed, maybe we have to check with other types again, possibly rechecking the subnodes. Maybe this gives exponential complexity. Note, that according to [2] there are better ways (with polynomial complexity).

3.2 An Upper Bound Complexity for Membership Test of Graph Shaped Data

The complexity is the same as in the tree case, *modulo* complexity for memoization.

3.3 An Upper Bound Complexity for Intersection of two NDFTA

Polynomial complexity, as we just have to build the cross product of both automata.

3.4 An Upper Bound Complexity for the Emptyness Test

```
* taversal of the graph ands seeking for an end-state-closed path
=> size of automaton
```

3.5 An Upper Bound Complexity for The Non-Deterministic Automaton Generated out of an R_2G_2 Instance

```
Cost of the single Rules:
* grammar : const. + rules
* rule : const. + re
* re : decompose
  * seq(r1,r2) : r1+r2+(1*epsilon)
  * disj(r1,r2) : r1+r2+(4*epsilon)
  * opt(r) : r+1*epsilon
  * plus(r) : r+1*epsilon
  * kleene(r) : r+1*epsilon
```

4 Acknowledgments.

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

References

[1] S. Berger and F. Bry. Towards Static Type Checking of Web Query Language. In *Proc. Workshop über Grundlagen von Datenbanken (GvD)*. 2005.

[2] H. Common, M. Dauchet, R. Gilleron, , F. J. D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata/`, 1999.

[3] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison Wesley 2006

[4] Benjamin C. Pierce. Types and Programming Languages. MIT Press 2002

[5] Sebastian Schaffert. Xcerpt: A Rule-Based Query and Transformation Language for the Web PhD Thesis, Institute for Informatics, University of Munich, 2004

[6] Uwe Schöning. Theoretische Informatik - kurzgefasst. Spektrum Akademischer Verlag, 2001