Collaborative Schema Construction using Regular Sequence Types

Jorge Coelho
ISEP & LIACC
Porto, Portugal
jcoelho@liacc.up.pt

Mário Florido
Univ. of Porto, DCC-FC & LIACC
Porto, Portugal
amf@dcc.fc.up.pt

Temur Kutsia
RISC, Johannes Kepler University
Linz, Austria
tkutsia@risc.uni-linz.ac.at

Abstract

In this paper we describe an approach to build XML schemas in a collaborative way. The approach is based on computing intersection between sequences of type terms built over constants, function symbols and type symbols. For type symbols there may be the corresponding type rules that specify the (regular) set of term sequences the type symbol generates. We describe the intersection algorithm, prove its termination and correctness, illustrate it on examples and show how it can be used in collaborative schema development.

1 Introduction

The work presented here is a new solution for the problem presented by the authors in [2]. In the previous work the authors proposed algorithms for Collaborative Schema Construction based on the unification of sequence variables. The new approach presented here is based on the intersection of regular sequence types.

Main idea behind *Collaborative Schema Construction* is the following: Several people are interested in producing a common schema for XML data and each of those people may impose some constraints on the schema structure. For example in the SuperJournal Project [11], the goal was *"to produce a cluster of journal content to make it worth the author submitting multimedia content and the reader doing useful searching and browsing in the electronic field with sufficient content that is relevant."*. SuperJournal brings a consortium of publishers to develop models for network publishing by gathering the content from several different journals. The consortium members propose a desired schema for their own domain. Collaboratively they could unify all their schemas in one general schema which satisfies everyone's requisites. After this all the information could be integrated in one general journal and distributed.

The integration of different schemas into a common, unified one, usually called *Schema Integration*, has been a prominent area of research for the database community over the past years [6, 9, 12]. With the widespread adoption of XML has the standard syntax to share data, new attention was given to schema integration for XML schemas [13, 8]. In all these previous works, schema integration means a semantic integration, i.e., in the different schemas, one may have different names for the same semantic concept, and this semantic knowledge is used to match schemas. Thus these works necessarily rely on domain specific information to perform matching. In our work, we assume that the syntax used in the different XML schemas is the same and that the several schemas are incomplete specifications where only some domain specific part is defined. Thus in this paper, when we refer to collaborative schema construction we mean *syntactic* collaborative schema construction.

The approach we follow here is to extend regular types [5] with sequences for typing XML documents and use type intersection on this extension in order to obtain a common schema. Our representation of regular types via sets of type rules, where each type rule has a symbol in the left hand side and a set of type term sequences in the right, makes implementation quite simple.

The rest of the paper is organized as follows: In Section 2 we present regular types and sequences and show how they are used together. The type intersection algorithm is presented in Section 3. Its application in Collaborative Schema Construction is demonstrated in Section 4. Finally, we conclude.

2 Types and Sequences

2.1 Terms and Regular Types

The next definitions and examples introduce briefly the notion of regular types along the lines presented in [5].

Definition 2.1 *Assuming an infinite set of* type symbols *and finite sets of* constants *and flexible arity* function symbols*, a* type term *is defined as follows:*

1. *A constant symbol is a type term.*

2. *A type symbol is a type term.*

3. *If f is a flexible arity function symbol and each τ_i is a type term, $f(\tau_1, \ldots, \tau_n)$ is a type term.*

We use a, b, c, d to denote constants, α, β, γ to denote type symbols and τ to denote type terms. A term *is a type term that does not contain a type symbol. We use t for terms and $\mathcal{T}$ for the set of all terms (over the given alphabet).*

Definition 2.2 *A* type rule *is an expression of the form $\alpha \rightarrow \Upsilon$ where α is a type symbol and Υ is a finite set of type terms.*

Example 2.1 *Let α and β be type symbols, $\alpha \rightarrow \{a, b\}$ and $\beta \rightarrow \{nil, tree(\beta, \alpha, \beta)\}$ are type rules.*

Definition 2.3 *A type symbol α is defined by a set of type rules $\mathcal{R}$ if there exists a type rule $\alpha \rightarrow \Upsilon \in \mathcal{R}$.*

We assume that there is at most one rule $\alpha \rightarrow \Upsilon$ in $\mathcal{R}$ for each α. We also assume that the set of type symbols contains μ (the universal type) and ϕ (the empty type) and no $\mathcal{R}$ contains rules for them.

Given a set of type rules $\mathcal{R}$ and a type term τ, the regular type $[\![\tau, \mathcal{R}]\!]$ (generated from τ by $\mathcal{R}$), is defined as the least set of terms satisfying the conditions:

- $[\![\tau, \mathcal{R}]\!] = \{a\}$, if τ is a constant a.
- $[\![\mu, \mathcal{R}]\!] = \mathcal{T}$ and $[\![\phi, \mathcal{R}]\!] = \emptyset$.
- $[\![\alpha, \mathcal{R}]\!] = \cup_{\alpha \rightarrow \Upsilon \in \mathcal{R}, \tau \in \Upsilon} [\![\tau, \mathcal{R}]\!]$, if τ is $\alpha \notin \{\mu, \phi\}$.

It is easy to see that our sets of type rules are equivalent to regular tree grammars [4] and types are regular tree languages.

2.2 Sequences and Sequence types

Below our considerations will focus on finite sequences of terms and extension of regular types to (regular) sequence types. We use $\tilde{t}$ for finite sequences of terms, $\tilde{\tau}$ for finite sequences of type terms and denote the empty sequence (of terms or type terms) with ϵ. For readability, sequences are written in parentheses, like, e.g., $(f(a, b), b, c)$ or $(f(\alpha, b), \beta, c)$.

Definition of type rules is extended to allow type term sequences instead of just type terms: A *type rule* is an expression of the form $\alpha \rightarrow \Upsilon$ where α is a type symbol and Υ is a finite set of *finite sequences of type terms*. Respectively, the definition of *sequence types* extends the one for types: Sequence types are sets of finite term sequences. This extension is obtained by introducing two additional conditions that deal with sequences: $[\![\epsilon, \mathcal{R}]\!] = \{\epsilon\}$ and $[\![(\tau, \tilde{\tau}), \mathcal{R}]\!] = \{(t, \tilde{t}) \mid t \in [\![\tau, \mathcal{R}]\!] \text{ and } \tilde{t} \in [\![\tilde{\tau}, \mathcal{R}]\!]\}$. Moreover, μ now generates the set of all finite sequences of terms (over the given alphabet).

Example 2.2 *If $\mathcal{R} = \{\alpha \rightarrow \{\epsilon, (f(c, \alpha), \alpha)\}$, then $[\![\alpha, \mathcal{R}]\!] = \{\epsilon, f(c), f(c, f(c)), \ldots, (f(c), f(c)), (f(c), f(c, f(c))), (f(c, f(c)), f(c)), \ldots, (f(c), f(c), f(c)), \ldots\}$.*

Note that extending regular types with sequences must be done together with some restrictions to avoid falling in the domain of context-free languages which are not closed under intersection and where testing for intersection emptiness is undecidable (two operations we need in our approach). Thus, we would like to forbid type rules sets like $\{\alpha \rightarrow \{\varepsilon, (a, \alpha, b)\}\}$ or like $\{\alpha \rightarrow \{\epsilon, (a, \beta, b)\}, \beta \rightarrow \{\alpha\}\}$. The restriction that we impose on the type rule sets, informally speaking, forbids cycles (in the type symbol dependency graph of the rules) via occurrences of type symbols in sequences on the topmost level, which are not the last sequence element occurrences. Examples of rule sets satisfying this restriction are: $\{\alpha \rightarrow \{\varepsilon, (a, \beta, b)\}, \beta \rightarrow \{f(\alpha)\}\}$ and $\{\alpha \rightarrow \{\varepsilon, (a, \beta, b)\}, \beta \rightarrow \{\epsilon, (c, \beta)\}\}$. Under this restriction, our types denote regular languages of term sequences (or regular hedge languages, in terminology of [10]).

An alternative approach to denote sequences of terms (trees, values) is to do it via regular expression types. See, e.g. [7]. regular operators are the symbols *, +, ?, | and the comma. For the given $\tilde{\tau}$, $\tilde{\tau}_1$ and $\tilde{\tau}_2$ the following table describes *regular expression types*:

$\tilde{\tau}*$	sequence of zero or more $\tilde{\tau}$'s
$\tilde{\tau}+$	sequence of one or more $\tilde{\tau}$'s
$\tilde{\tau}?$	zero or one $\tilde{\tau}$
$\tilde{\tau}_1 \mid \tilde{\tau}_2$	$\tilde{\tau}_1$ or $\tilde{\tau}_2$
$\tilde{\tau}_1, \tilde{\tau}_2$	$\tilde{\tau}_1$ followed by $\tilde{\tau}_2$.

We translate regular expression types to our notation. The translation is made accordingly to the following rules:

$$
\begin{array}{lcl}
\tilde{\tau}* & \Rightarrow & \alpha_* \rightarrow \{\epsilon, (\tilde{\tau}, \alpha_*)\} \\
\tilde{\tau}+ & \Rightarrow & \alpha_+ \rightarrow \{\tilde{\tau}, (\tilde{\tau}, \alpha_+)\} \\
\tilde{\tau}? & \Rightarrow & \alpha_? \rightarrow \{\epsilon, \tilde{\tau}\} \\
\tilde{\tau}_1 \mid \tilde{\tau}_2 & \Rightarrow & \alpha_| \rightarrow \{\tilde{\tau}_1, \tilde{\tau}_2\} \\
\tilde{\tau}_1, \tilde{\tau}_2 & \Rightarrow & \alpha_{\text{seq}} \rightarrow \{(\tilde{\tau}_1, \tilde{\tau}_2)\}
\end{array}
$$

3 Type intersection

The intersection of types is the core of our new approach to Collaborative Schema Construction. In this section we present some definitions, the intersection algorithm, correctness results and examples.

We start with a function *empty*, that operates on a sequence of type terms $\tilde{\tau}$ and a set of rules $\mathcal{R}$ and decides whether $[\![\tilde{\tau}, \mathcal{R}]\!] = \emptyset$. The function keeps a trail S of tried

type symbols and is defined as follows:

$$
\begin{aligned}
\mathit{empty}(\tilde{\tau}, \mathcal{R}) &= \mathit{empty}(\tilde{\tau}, \mathcal{R}, \emptyset).\\
\mathit{empty}(\tilde{\tau}, \mathcal{R}, S) &= \mathit{False}, \text{ if } \tilde{\tau} \in \{\epsilon, \mu\}\\
\mathit{empty}((\tau, \tilde{\tau}), \mathcal{R}, S) &= \mathit{empty}(\tau, \mathcal{R}, S) \wedge \mathit{empty}(\tilde{\tau}, \mathcal{R}, S).\\
\mathit{empty}(\phi, \mathcal{R}, S) &= \mathit{True}.\\
\mathit{empty}(a, \mathcal{R}, S) &= \mathit{False}.\\
\mathit{empty}(f(\tilde{\tau}), \mathcal{R}, S) &= \mathit{empty}(\tilde{\tau}, \mathcal{R}, S).\\
\mathit{empty}(\alpha, \mathcal{R}, S) &= \mathit{True} \text{ if } \alpha \in S.\\
\mathit{empty}(\alpha, \mathcal{R}, S) &= \wedge_{i=1}^{n} \mathit{empty}(\tilde{\tau}_i, \mathcal{R}, S \cup \{\alpha\}),\\
&\quad \text{if } \alpha \to \{\tilde{\tau}_1, \ldots, \tilde{\tau}_n\} \in \mathcal{R}, \alpha \notin S.
\end{aligned}
$$

The function *empty* is correct: If $[\![\tilde{\tau}, \mathcal{R}]\!] = \emptyset$, then $\mathit{empty}(\tilde{\tau}, \mathcal{R}) = \mathit{True}$, otherwise $\mathit{empty}(\tilde{\tau}, \mathcal{R}) = \mathit{False}$. It follows from the correctness of a similar function applied to general regular types in [14].

Example 3.1 $\mathit{empty}(\alpha, \{\alpha \to \{(a, \alpha)\}) = \mathit{True}$.

Another useful function is $\mathit{nullable}$. Given a sequence of type terms $\tilde{\tau}$ and a set of type rules $\mathcal{R}$, the function decides whether $\epsilon \in [\![\tilde{\tau}, \mathcal{R}]\!]$. It keeps a trail S of tried type symbols and is defined as follows:

$$
\begin{aligned}
\mathit{nullable}(\tilde{\tau}, \mathcal{R}) &= \mathit{nullable}(\tilde{\tau}, \mathcal{R}, \emptyset).\\
\mathit{nullable}(\epsilon, \mathcal{R}, S) &= \mathit{True}.\\
\mathit{nullable}((\tau, \tilde{\tau}), \mathcal{R}, S) &= \mathit{nullable}(\tau, \mathcal{R}, S) \wedge\\
&\quad \mathit{nullable}(\tilde{\tau}, \mathcal{R}, S).\\
\mathit{nullable}(\mu, \mathcal{R}, S) &= \mathit{True}.\\
\mathit{nullable}(\phi, \mathcal{R}, S) &= \mathit{False}.\\
\mathit{nullable}(a, \mathcal{R}, S) &= \mathit{False}.\\
\mathit{nullable}(f(\tilde{\tau}), \mathcal{R}, S) &= \mathit{False}.\\
\mathit{nullable}(\alpha, \mathcal{R}, S) &= \mathit{False} \text{ if } \alpha \in S.\\
\mathit{nullable}(\alpha, \mathcal{R}, S) &= \vee_{i=1}^{n} \mathit{nullable}(\tilde{\tau}_i, \mathcal{R}, S \cup \{\alpha\}),\\
&\quad \text{if } \alpha \to \{\tilde{\tau}_1, \ldots, \tilde{\tau}_n\} \in \mathcal{R}, \alpha \notin S.
\end{aligned}
$$

It is not hard to show that $\mathit{nullable}$ is correct: If $\epsilon \in [\![\tilde{\tau}, \mathcal{R}]\!]$, then $\mathit{nullable}(\tilde{\tau}, \mathcal{R}) = \mathit{True}$, otherwise $\mathit{nullable}(\tilde{\tau}, \mathcal{R}) = \mathit{False}$.

We now present the sequence type intersection algorithm. Given a set of type rules $\mathcal{R}_{\text{in}}$ and two sequences of type terms $\tilde{\tau}_1$ and $\tilde{\tau}_2$, the algorithm returns a sequences of type terms $\tilde{\tau}$ and a set of type rules $\mathcal{R}_{\text{out}}$ such that $[\![\tilde{\tau}, \mathcal{R}_{\text{out}}]\!] = [\![\tilde{\tau}_1, \mathcal{R}_{\text{in}}]\!] \cap [\![\tilde{\tau}_2, \mathcal{R}_{\text{in}}]\!]$.

Algorithm 1 shows the rules. We use two global variables, R for the type rules and T for the trail that stores the intersections already made in order to avoid cycles. Whenever more than one case is applicable the first one is used. The rules are formulated modulo commutativity of intersection. The letter ρ in the cases 5 and 7 stands for a type term that is not a type symbol.

Algorithm 1: Intersection of Sequences of Type Terms

Input: $\tilde{\tau}_1$, $\tilde{\tau}_2$ and a set of type rules $\mathcal{R}_{\text{in}}$. R is a global variable initialized by $\mathcal{R}_{\text{in}}$. T is a global variable initialized by $\emptyset$. The algorithm returns either ϕ or a sequence of type terms $\tilde{\tau}$ together with the set of type rules $\mathcal{R}_{\text{out}}$ which is the value assigned to of the global variable R. One of the following cases apply depending on the shape of $\tilde{\tau}_1$ and $\tilde{\tau}_2$.

1. $\tilde{\tau} \cap \tilde{\tau} = \tilde{\tau}$.
2. $\tilde{\tau}_1 \cap \tilde{\tau}_2 = \tilde{\tau}$, if $(\tilde{\tau}_1, \tilde{\tau}_2, \tilde{\tau}) \in T$ or $(\tilde{\tau}_2, \tilde{\tau}_1, \tilde{\tau}) \in T$.
3. $\tilde{\tau} \cap \mu = \tilde{\tau}$.
4. $f(\tilde{\tau}_1) \cap f(\tilde{\tau}_2) = f(\tilde{\tau}_1 \cap \tilde{\tau}_2)$.
5. $(\rho_1, \tilde{\tau}_1) \cap (\rho_2, \tilde{\tau}_2) = (\rho_1 \cap \rho_2, \tilde{\tau}_1 \cap \tilde{\tau}_2)$.
6. $\tilde{\tau} \cap \epsilon = \epsilon$, if $\mathit{nullable}(\tilde{\tau}, R) = \mathit{True}$.
7. $(\alpha, \tilde{\tau}_1) \cap (\rho, \tilde{\tau}_2) = \gamma$, where γ is a new type symbol. T and R are updated as follows:
   $T := T \cup \{(\alpha, \tilde{\tau}_1), (\rho, \tilde{\tau}_2), \gamma\}$;
   Select $\alpha \to \Upsilon_1 \in R$;
   $\Upsilon := \emptyset$;
   foreach $\tilde{\tau}_1' \in \Upsilon_1$ **do**
   $\quad \tilde{\tau}' := (\tilde{\tau}_1', \tilde{\tau}_1) \cap (\rho, \tilde{\tau}_2)$;
   $\quad$ **if** $\mathit{empty}(\tilde{\tau}', R) = \mathit{False}$ **then** $\Upsilon := \Upsilon \cup \{\tilde{\tau}'\}$
   if $\Upsilon \neq \emptyset$ **then**
   $\quad R := R \cup \{\gamma \to \Upsilon\}$;
   else
   $\quad$ Failure: this case is not applicable.
8. $(\alpha, \tilde{\tau}_1) \cap (\beta, \tilde{\tau}_2) = \gamma$, where γ is a new type symbol. T and R are updated as follows:
   $T := T \cup \{(\alpha, \tilde{\tau}_1), (\beta, \tilde{\tau}_2), \gamma\}$;
   Select $\alpha \to \Upsilon_1 \in R$ and $\beta \to \Upsilon_2 \in R$;
   $\Upsilon := \emptyset$;
   foreach $\tilde{\tau}_1' \in \Upsilon_1$ *and* $\tilde{\tau}_2' \in \Upsilon_2$ **do**
   $\quad \tilde{\tau}' := (\tilde{\tau}_1', \tilde{\tau}_1) \cap (\tilde{\tau}_2', \tilde{\tau}_2)$;
   $\quad$ **if** $\mathit{empty}(\tilde{\tau}', R) = \mathit{False}$ **then** $\Upsilon := \Upsilon \cup \{\tilde{\tau}'\}$
   if $\Upsilon \neq \emptyset$ **then**
   $\quad R := R \cup \{\gamma \to \Upsilon\}$;
   else
   $\quad$ Failure: this case is not applicable.
9. $\tilde{\tau}_1 \cap \tilde{\tau}_2 = \phi$, if none of the previous case applies.

Example 3.2 *Let* $\mathcal{R}_{\text{in}}$ *be* $\{\alpha \to \{\epsilon, (a, \alpha)\}\}$. *We want to compute* $\alpha \cap (a, \alpha)$. *The case 7 applies and the result is computed as follows:* $\epsilon \cap (a, \alpha) = \phi$ *by case 9,* $(a, \alpha) \cap (a, \alpha) = (a, \alpha)$ *by case 1. Hence, the type symbol* γ *denotes the in-*

tersection with $\mathcal{R}_{\text{out}} = \{\alpha \to \{\epsilon, (a, \alpha)\}, \gamma \to \{(a, \alpha)\}\}$.

Example 3.3 *Let* $\mathcal{R}_{\text{in}}$ *be* $\{\alpha \to \{\epsilon, (a, \alpha)\}, \beta \to \{a, (a, \beta)\}\}$ *and compute* $\alpha \cap \beta$. *The case 8 applies,* T *becomes* $\{(\alpha, \beta, \gamma)\}$ *and the result is computed as follows:* $\epsilon \cap a = \phi$ *by case 9;* $\epsilon \cap (a, \beta) = \phi$ *by case 9;* $(a, \alpha) \cap a = a$ *by case 5 (because* $a \cap a = a$, $\alpha \cap \epsilon = \epsilon$ *and* (a, ϵ) *is a); and, finally,* $(a, \alpha) \cap (a, \beta)$ *by case 5 becomes* $(a, \alpha \cap \beta)$ *and we should compute* $\alpha \cap \beta$, *but since* $(\alpha, \beta, \gamma) \in T$, *the case 2 gives* γ *on that. Hence,* $\Upsilon = \{a, (a, \gamma)\}$ *and the result is* γ *together with* $\mathcal{R}_{\text{out}} = \{\alpha \to \{\epsilon, (a, \alpha)\}, \beta \to \{a, (a, \beta)\}, \gamma \to \{a, (a, \gamma)\}\}$.

Example 3.4 *Let* $\mathcal{R}_{\text{in}}$ *be* $\{\alpha \to \{\alpha\}, \beta \to \{\beta\}\}$. *To compute* $\alpha \cap \beta$, *we try to apply the transformation in case 8. It introduces a new type symbol* γ, *adds* (α, β, γ) *to* T *and proceeds with computing again* $\alpha \cap \beta$. *Now, the case 2 applies and we get* γ. *But* $R = \mathcal{R}_{\text{in}}$ *does not contain a rule for* γ. *Therefore,* $empty(\gamma, R) = True$ *and* Υ *in case 8 remains empty. Therefore, the attempt of applying case 8 fails and the case 9 finally gives* $\alpha \cap \beta = \phi$.

Example 3.5 *Let* $\mathcal{R}_{\text{in}}$ *consist of the following type rules:*

$$\begin{array}{lcl} \alpha_1 & \to & \{\epsilon, (a, \alpha_1)\} \\ \alpha_2 & \to & \{(b, \alpha_3, \alpha_4)\} \\ \alpha_3 & \to & \{c, (c, \alpha_3)\} \\ \alpha_4 & \to & \{\epsilon, (d, \alpha_4)\} \\ \alpha_5 & \to & \{(b, \alpha_6)\} \\ \alpha_6 & \to & \{\epsilon, (c, \alpha_6)\} \end{array}$$

and compute the intersection $l(\alpha_1, \alpha_2) \cap l(a, a, \alpha_5)$. *Note that these types can be seen as the following XML types:*

```
<!ELEMENT l(a*,b,c+,d*)>
<!ELEMENT a (#PCDATA)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>
<!ELEMENT d (#PCDATA)>

<!ELEMENT l(a,a,b,c*)>
<!ELEMENT a (#PCDATA)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>
```

We go through this computation stepwise, but for simplicity omit some steps. By case 4 we have that,

$$l(\alpha_1, \alpha_2) \cap l(a, a, \alpha_5) = l((\alpha_1, \alpha_2) \cap (a, a, \alpha_5))$$

Thus, we should compute $(\alpha_1, \alpha_2) \cap (a, a, \alpha_5)$, *which, by case 7, introduces a new type symbol* γ_1 *and computes further the following two intersections:*

1. $\epsilon \cap (a, a, \alpha_5)$ *which results in* ϕ.
2. $(a, \alpha_1, \alpha_2) \cap (a, a, \alpha_5)$ *which by cases 5 and 1 results in* $(a, (\alpha_1, \alpha_2) \cap (a, \alpha_5))$. *For* $(\alpha_1, \alpha_2) \cap (a, \alpha_5))$, *by case 7 a new type symbol* γ_2 *is introduced and the following intersection are to be computed:*

2.1. $\alpha_2 \cap (a, \alpha_5)$, *which results in* ϕ.

2.2. $(a, \alpha_1, \alpha_2) \cap (a, \alpha_5)$ *which by cases 5 and 1 results in* $(a, (\alpha_1, \alpha_2) \cap \alpha_5)$. *By case 8 a new type symbol* γ_3 *the following intersections are to be computed:*

2.2.1. $(a, \alpha_1, \alpha_2) \cap (b, \alpha_6)$, *which leads to* ϕ.

2.2.2. $\alpha_2 \cap (b, \alpha_6)$, *which by case 7 introduces a new type symbol* γ_4 *and* $(b, \alpha_3, \alpha_4) \cap (b, \alpha_6)$ *is to be computed. By case 5 it results in* $(b, (\alpha_3, \alpha_4) \cap \alpha_6)$. *Thus, we compute* $(\alpha_3, \alpha_4) \cap \alpha_6$. *By case 8 a new type symbol* γ_5 *is introduced and* $(c, \alpha_4) \cap \alpha_6$ *and* $(c, \alpha_3, \alpha_4) \cap \alpha_6$ *should be computed. The first one eventually leads to* c *as the result, while the second one reduces to computing* $(\alpha_3, \alpha_4) \cap \alpha_6$. *A new type symbol* γ_6 *is introduced. It is important to not that a triple* $((\alpha_3, \alpha_4), \alpha_6, \gamma_6)$ *is stored in* T. *We now have:* $(c, \alpha_4) \cap \alpha_6$ *that gives* c *as we have already seen, and* $(c, \alpha_3, \alpha_4) \cap \alpha_6$, *that returns by case 2* γ_6 *because the corresponding triple is already in* T.

Now we look what is our $\mathcal{R}_{\text{out}}$: *This is* $\mathcal{R}_{\text{in}} \cup \{\gamma_6 \to \{c, (c, \gamma_6)\}, \gamma_5 \to \{c, (c, \gamma_6)\}, \gamma_4 \to \{(b, \gamma_5)\}, \gamma_3 \to \{\gamma_4\}, \gamma_2 \to \{(a, \gamma_3)\}, \gamma_1 \to \{(a, \gamma_2)\}\}$. *This set and* $l(\gamma_1)$ *are the output of the algorithm.*

We can simplify the rules, getting the result $l(\gamma_1)$ *with* $\gamma_1 \to \{(a, a, b, \gamma_5)\}$ *and* $\gamma_5 \to \{c, (c, \gamma_5)\}$. *It can be seen as a representation of the following DTD:*

```
<!ELEMENT l(a,a,b,c+)>
<!ELEMENT a (#PCDATA)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>
```

Now we prove correctness of the algorithm. First, ne need the following lemma:

Lemma 3.1 *Let* $\tilde{\tau}_1$ *and* $\tilde{\tau}_2$ *be sequences of type terms and* $\mathcal{R}$ *be a set of type rules such that each type symbol in* $\tilde{\tau}_1$, $\tilde{\tau}_2$ *and* $\mathcal{R}$ *has exactly one defining rule in* $\mathcal{R}$. *If the intersection algorithm for* $\tilde{\tau}_1 \cap \tilde{\tau}_2$ *and* $\mathcal{R}_{\text{in}} = \mathcal{R}$ *returns* $\tilde{\tau}$ *and the set of type rules* $\mathcal{R}_{\text{out}}$, *then each type symbol in* $\tilde{\tau}$ *and* $\mathcal{R}_{\text{out}}$ *has exactly one defining rule in* $\mathcal{R}_{\text{out}}$.

Proof 3.1 *It is straightforward to see that the result of intersection is always a sequence of type terms. New type symbols are only introduced in cases 7 and8 where a new type rule for this new type symbol is introduced. For each new symbol there is only one new type rule. Then the result follows by induction on the number of recursive calls to the intersection.*

Theorem 3.1 (Correctness) *Let $\tilde{\tau}_1$ and $\tilde{\tau}_2$ be sequences of type terms and $\mathcal{R}$ be the corresponding set of type rules. Then the intersection algorithm for $\tilde{\tau}_1 \cap \tilde{\tau}_2$ and $\mathcal{R}_{\rm in} = \mathcal{R}$ terminates and returns $\tilde{\tau}$ and $\mathcal{R}_{\rm out}$ such that $[\![\tilde{\tau}, \mathcal{R}_{\rm out}]\!] = [\![\tilde{\tau}_1, \mathcal{R}_{\rm in}]\!] \cap [\![\tilde{\tau}_2, \mathcal{R}_{\rm in}]\!]$.*

Proof 3.2 *We prove the theorem in two steps:*

Termination. *During computation, the set T never contains two triples $(\tilde{\tau}'_1, \tilde{\tau}'_2, \gamma')$ $(\tilde{\tau}''_1, \tilde{\tau}''_2, \gamma'')$ such that $\tilde{\tau}'_1 = \tilde{\tau}''_1$ and $\tilde{\tau}'_2 = \tilde{\tau}''_2$. From the cases 6, 7 and 8 it follows that if a triple $(\tilde{\tau}'_1, \tilde{\tau}'_2, \gamma')$ is in T, then both $\tilde{\tau}'_1$ and $\tilde{\tau}'_2$ belong to the finite set S, which consists of subexpressions of the initial $\tilde{\tau}_1$ and $\tilde{\tau}_2$, and of subexpressions of expressions occurring in Υ in rules $\alpha \to \Upsilon \in \mathcal{R}_{\rm in}$. Hence, the set $T' = \{(\tilde{\tau}_1, \tilde{\tau}_2) \mid (\tilde{\tau}'_1, \tilde{\tau}'_2, \gamma') \in T\}$ is finite. Let $\bar{T}'$ be the complement of T' to the (finite) set $\{(\tilde{\tau}'_1, \tilde{\tau}'_2) \mid \tilde{\tau}'_1 \in S, \tilde{\tau}_2 \in S\}$.*

By $|\tilde{\tau}|$ we denote the size of $\tilde{\tau}$, i.e., its denotational length. With each intersection problem $\tilde{\tau}_1 \cap \tilde{\tau}_2$ and the corresponding value of T in the algorithm we associate a pair $(\bar{T}', |\tilde{\tau}_1| + |\tilde{\tau}_2| + 1)$, calling it the complexity measure of the state $(\tilde{\tau}_1, \tilde{\tau}_2, T)$ of the algorithm. The measures are compared lexicographically: The first arguments are compared by set inclusion, the second ones - by the standard ordering on natural numbers. We denote this ordering by $<$ and note that it is well-founded.

First, note that if the cases 1–3, 5, and 9 are applied, the intersection algorithm terminates immediately. In the other cases we compares the complexity measures of states at two successive calls of the algorithm. The cases 4 and 5 do not change the first component of the measure, but decrease the second one strictly. The other two cases strictly decrease the first component.

Hence, each case of the algorithm either terminates immediately, or makes a recursive call with strictly decreased complexity measure. Since the ordering is well-founded, the algorithm terminates.

Correctness *We should show that the transformation in each case of the algorithm is sound. That is, if the case applies to $\tilde{\tau}_1 \cap \tilde{\tau}_2$ together with R', and the result of application is $\tilde{\tau}$ together with R'', then $[\![\tilde{\tau}, R'']\!] = [\![\tilde{\tau}_1, R']\!] \cap [\![\tilde{\tau}_2, R']\!]$. For the cases 1, 2 and 3 it is straightforward. For the cases 4, 5 and 6 it follows from the definition of types and basic properties of sequence intersection. For the cases 7 and 8 one needs the definition of types and basic properties of union and (sequence) intersection. Since these rules exhaust all potentially successful transformations, if none of them are applicable then the intersection should be empty. It justifies the case 9.*

4 Application in Collaborative Schema Construction

We now show how to apply the regular types intersection to Collaborative Schema Construction. The idea is that several people are interested in producing a common schema for XML data and each of those people may impose some constraints on the schema structure.

We now demonstrate our approach with an example:

Example 4.1 *Three entities need to share a custom address book document. All of them agree that the document should have root element* addrbook *and one element* name*, and each of them imposes other constraints in the document content (note that places representing sequences which are not of interest for a given entity are represented by an underscore '_'):*

1. *The document must have a sequence of zero or more elements* address *after* name *and does not matter what comes next. The proposed document has the simplified structure: $addrbook(name, address*, \_)$.*
2. *The document must have a sequence of one or more elements* email *somewhere, with the simplified structure: $addrbook(name, \_, email+, \_)$*
3. *The document must have one or more* address *elements after* name *and an optional* telephone *element somewhere, with the following simplified structure: $addrbook(name, address+, \_, telephone?, \_)$*

These requirement can be described by the following types rules (note that the underscores are translated to the universal type represented by μ):

$$\begin{array}{lcl} \alpha_1 & \to & \{addrbook(name, \alpha_2, \mu)\} \\ \alpha_2 & \to & \{\epsilon, (address, \alpha_2)\} \\ \alpha_3 & \to & \{addrbook(name, \mu, \alpha_4, \mu)\} \\ \alpha_4 & \to & \{email, (email, \alpha_4)\} \\ \alpha_5 & \to & \{addrbook(name, \alpha_6, \mu, \alpha_7, \mu)\} \\ \alpha_6 & \to & \{address, (address, \alpha_6)\} \\ \alpha_7 & \to & \{\epsilon, telephone\} \end{array}$$

The idea is to calculate the intersection between α_1, α_3 and α_5. We start by calculating the intersection between α_1 and α_3 and we will intersect the result with α_5. We omit some steps for the sake of clarity.

We start by case 8 of the intersection algorithm. It introduces a new type symbol γ_1 and calls to compute:

- *$addrbook(name, \alpha_2, \mu) \cap addrbook(name, \mu, \alpha_4, \mu)$. It reduces to $(\alpha_2, \mu) \cap (\mu, \alpha_4, \mu))$. From here, $\alpha_2 \cap \mu = \alpha_2$ and $\mu \cap (\alpha_4, \mu) = (\alpha_4, \mu)$. Thus we have,*

$$\gamma_1 \to \{addrbook(name, \alpha_2, \alpha_4, \mu)\}.$$

Now we compute $\gamma_1 \cap \alpha_5$. First, a new type symbol γ_2 is introduced and we get to $addrbook(name, (\alpha_2, \alpha_4, \mu) \cap (\alpha_6, \mu, \alpha_7)$. This results, on the one hand, in the intersection of $\alpha_2 \cap \alpha_6$ which results in a new type symbol γ_3 with the rule $\gamma_3 \rightarrow \{address, (address, \epsilon)\}$. On the other hand, we need to compute the intersection $(\alpha_4, \mu) \cap (\mu, \alpha_7, \mu)$. From here, we get $\alpha_4 \cap \mu = \alpha_4$ and $\mu \cap (\alpha_7, \mu) = (\alpha_7, \mu)$. Hence,

$$\begin{array}{lcl} \gamma_2 & \rightarrow & \{addrbook(name, \gamma_3, \alpha_4, \alpha_7, \mu)\} \\ \gamma_3 & \rightarrow & \{address, (address, \gamma_3)\} \\ \alpha_4 & \rightarrow & \{email, (email, \alpha_4)\} \\ \alpha_7 & \rightarrow & \{\epsilon, telephone\} \end{array}$$

Now we can drop the universal type present on the tail since it has no further utility. The resulting type rules are trivially translated to the following DTD:

```
<!ELEMENT addrbook (name,address+,
                    email+,telephone?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

5 Conclusion and Future Work

In this paper we presented a new approach to Collaborative Schema Construction. It is based on computing intersection between sequences of type terms. We believe that this new approach is simpler than the previous presented by the authors in [2].

The implementation of these algorithms in the language XCentric is based on the introduction of a special operator and is described in the report available at: [3].

Our future work will be centered on continuing the integration of these techniques in the XML-processing logic language XCentric [1].

References

[1] J. Coelho and M. Florido. XCentric: logic programming for XML processing. In I. Fundulaki and N. Polyzotis, editors, *WIDM*, pages 1–8. ACM, 2007.

[2] J. Coelho, M. Florido, and T. Kutsia. Sequence disunification and its application in collaborative schema construction. In M. Weske, M.-S. Hacid, and C. Godart, editors, *WISE Workshops*, volume 4832 of *Lecture Notes in Computer Science*, pages 91–102. Springer, 2007.

[3] J. Coelho, M. Florido, and T. Kutsia. Collaborative schema construction using regular types: Implementation. Technical report, University of Porto, `http://www.liacc.up.pt/~jcoelho/CollaborativeRTImpl.pdf`, 2009.

[4] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available from: `http://www.grappa.univ-lille3.fr/tata`, 2007.

[5] P. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*. The MIT Press, 1992.

[6] A. Doan and A. Y. Halevy. Semantic integration research in the database community: A brief survey. *AI Magazine*, 26(1):83–94, 2005.

[7] H. Hosoya and B. Pierce. Regular expression pattern matching for XML. *J. Functional Programming*, 13(6):961–1004, 2003.

[8] D. Kensche, C. Quix, M. A. Chatti, and M. Jarke. Gerome: A generic role based metamodel for model management. *J. Data Semantics*, 8:82–117, 2007.

[9] Y. Lee, M. Sayyadian, A. Doan, and A. Rosenthal. eTuner: tuning schema matching software using synthetic scenarios. *VLDB J.*, 16(1):97–122, 2007.

[10] M. Murata. Extended path expressions for XML. In *PODS*. ACM, 2001.

[11] D. J. Pullinger. *SuperJournal Project*. IOP Publishing Ltd., Bristol, UK, 1994.

[12] C. Quix, D. Kensche, and X. Li. Generic schema merging. In *CAiSE*, pages 127–141, 2007.

[13] E. Rahm, H. Hai-Do, and S. Massmann. Matching large XML schemas. *SIGMOD Rec.*, 33(4):26–31, 2004.

[14] J. Zobel. *Analysis of Logic Programs*. PhD thesis, Department of Computer Science, University of Melbourne, 1990.