

Charles University in Prague
Faculty of Mathematics and Physics

HABILITATION THESIS



RNDr. Irena Holubová, Ph.D.

**Evolution and Adaptability
of Complex XML Applications**

Prague, November 2012

Evolution and Adaptability of Complex XML Applications

Habilitation thesis

Irena Holubová (Mlýnková)

November, 2012

`holubova@ksi.mff.cuni.cz`

`http://www.ksi.mff.cuni.cz/~holubova/index.html`

Charles University in Prague
Faculty of Mathematics and Physics
Department of Software Engineering
Malostranské nám. 25
118 00, Prague 1
Czech Republic

This thesis contains copyrighted material. The copyright holders are:

©Springer-Verlag
©Elsevier B.V.
©Oxford University Press
©IOS Press

Acknowledgments

Since the very nature of scientific work is cooperation, I would like to thank especially to all my colleagues and co-authors. In the first place my thanks go to members and Ph.D. students of the *XML and Web Engineering Research Group* (XRG), namely to Martin Nečaský, Jaroslav Pokorný, Karel Richta, Jakub Klímek, Tomáš Knap, Jakub Malý, Jakub Stárka, and Martin Svoboda, as well as all the excellent students whose Master theses and SW projects form significant parts of our research results. Secondly, I am very grateful for the opportunity to visit during my research stay Dr. Eric Pardede from the La Trobe University, Melbourne and Dr. Sherif Sakr from the NICTA, Sydney and to get acquainted with their inspiring work and research topics. For helpful comments, suggestions, and ideas that enabled to improve our results I must undoubtedly express my gratitude to all the anonymous reviewers of our papers, audience at our presentations, and all the scientists I had the pleasure to meet and talk to. And, last but not least, I am very thankful to my family whose support and understanding enables me to do the job I like.

Contents

1	Commentary	1
1.1	Five-Level Evolution Management Framework	6
1.1.1	XML View	10
1.2	Adaptation of XML Data	12
1.3	Reverse Engineering: Inference of XML Schemas	15
1.4	Reverse Engineering: Mapping between Schemas	18
1.5	Analysis of Real-World Data	19
1.6	Author's Contributions	21
2	Evolution and Change Management of XML-Based Systems	23
3	Efficient Adaptation of XML Data Using a Conceptual Model	51
4	Heuristic Methods for Inference of XML Schemas: Lessons Learned and Open Issues	87
5	Structural and Semantic Aspects of Similarity of Document Type Definitions and XML Schemas	129
6	Analyzer: A Complex System for Data Analysis	151
7	Conclusion	179
7.1	Current and Future Research	180

Preface

The proposed thesis presents selected results of the author's research in the area of evolution of XML applications. The research has been carried out at the Faculty of Mathematics and Physics of the Charles University in Prague in years 2007–2012, mainly within the *XML and Web Engineering Research Group* (XRG)¹ lead by Prof. RNDr. Jaroslav Pokorný, CSc.

The results are presented as a collection of five selected papers. Four of them [24, 30, 34, 45] fit into a single framework, whereas each focuses on a particular subproblem. The last included paper [41] describes a separate, but closely related topic. The papers are presented in separate chapters (2–6) in their camera-ready forms (of *the International Journal of Systems and Software*, *the International Journal on Information Systems Frontiers*, *Informatica – the International Journal*, *the International Journal on Information Sciences*, and *the Computer Journal*), whereas the unifying commentary is provided in Chapter 1. Prior to summarizing the papers, the commentary provides a motivation and briefly surveys related state-of-the-art results. In order to provide a quick navigation, the references to the author's original contributions are marked with a star (see on the right). In Chapter 7 we conclude and outline directions of our current and future research. 

The research included in the selected papers has been supported by several grants, namely Information Society 1ET100300419, GAČR 201/06/0756, GAČR 201/09/0990, GAČR 201/09/P364, GAČR P202/10/0573, and TAČR TA02010182.

Prague, November 2012

Irena Holubová

¹<http://www.ksi.mff.cuni.cz/xrg/>

Chapter 1

Commentary

The most common application of the today's information-technology (IT) world are *information systems*. They can be characterized as a network of software and hardware components that enable people and companies to create, collect, distribute and process data. There exist various types of information systems, such as, e.g., decision-support systems, database-management systems, or office-information systems. Currently a very popular kind of information system are distributed information systems. For instance, such an information system can be based on the *Service Oriented Architecture* (SOA) [1] and its most common implementation – *Web Services* [8].

An information system often involves a huge set of *data resources* and a set of sub-applications, each being responsible for a particular logical execution part (we speak about a *system of applications*). A data resource consists of the particular pieces of information (i.e., *data instances*), *integrity constraints* over the data instances, and selected *interpretations* of the data. For instance, an information system for storing and analysis of scientific publications of academic institutions involves records of publications of a particular university, i.e., articles, books, SW prototypes, etc. Over the records we can identify various integrity constraints, such as, e.g., that “each book must be assigned with a unique ISBN” or that “a publication cannot have two authors with the same name”. Finally, for the purpose of presentation of the results at web sites of the institutions, the data may be exported in XML [7] or HTML [2], described in WSDL [6], exchanged using SOAP [4] messages, etc.

The life cycle of a complex system of applications is similar to a life cycle of a single application; however, the complexity is much higher. First of all,

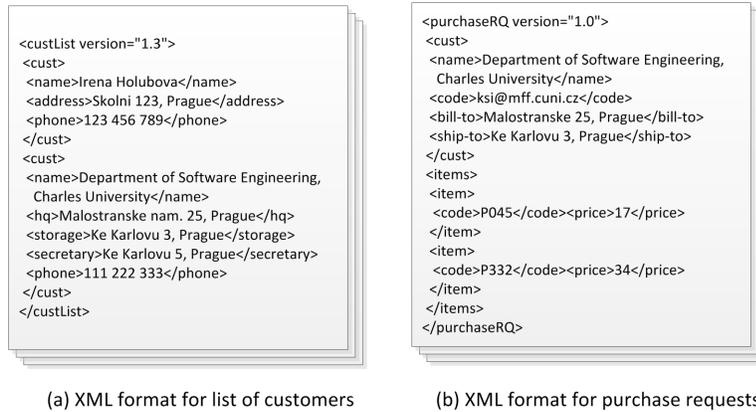


Figure 1.1: Sample XML documents from the same problem domain

we need to design numerous data structures, i.e., schemas, which are usually mutually related or overlaid. In other words, each application of the system utilizes several *views* of a common problem domain. Hence, they cannot be designed separately. In addition, sooner or later the user requirements of the applications change and, hence, the data structures they process must be modified respectively – we speak about the problem of *evolution*. Due to the relations and overlays, such a modification can influence multiple parts of the system and we need to maintain them during the whole life cycle to be able to propagate the changes completely, correctly and efficiently. The ability of an information system to adapt to the changes is called *adaptability*.

Simple Motivating Example Let us consider a company that receives purchase orders and let us focus on the part of the system that processes purchases. Let the messages used in the system be XML messages formatted according to a family of different XML formats. Consider the two sample XML documents in Figure 1.1. The former one is formatted according to an XML format for a list of customers. The latter one is formatted according to a different XML format for purchase requests. As we can see, the concept of *customer* is represented in each of our sample XML formats in a different way (i.e., viewed from different perspectives). In the first one, different kinds of customers are distinguished (private and corporate customers). For private customers, elements **name**, **address** and **phone** are present. For corporate customers, elements **name**, different addresses (**hq**, i.e., headquarters, **storage** and **secretary**), and **phone** are present. In the second XML doc-

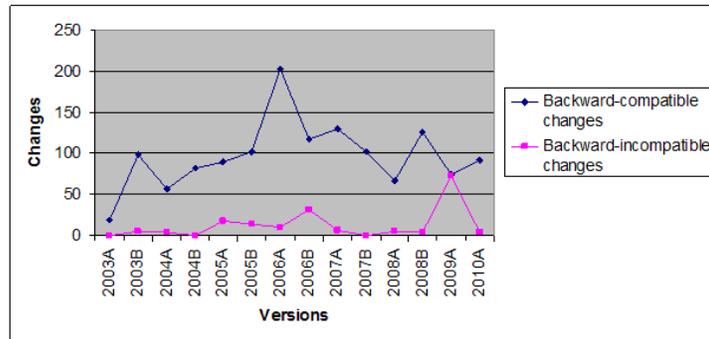


Figure 1.2: OpenTravel.org schema changes in 2003–2010

ument, we do not distinguish different types of customers. We have only element `cust` with child elements `name`, `code`, `ship-to` and `bill-to`. The last two represent addresses. However, this is different representation than in the previous sample. We need a unified representation of shipping and billing addresses in purchase requests for all kinds of customers.

Let us now consider a new user requirement that an address should no longer be represented as a simple string. Instead, it should be divided into elements `street`, `city`, `zip`, etc. Apparently, in a complex system comprising tens or even hundreds of XML formats, this is a difficult and error-prone task. Even identifying the affected parts is not an easy and straightforward process. For example, we may need to make the modification only for addresses that represent a place where to ship the goods (i.e., elements `address` and `storage` in the XML format instantiated in the first schema and element `ship-to` in the second schema), whereas we do not want to modify addresses that represent headquarters, etc.

Real-World Motivating Example A possible solution to the problem of evolution adopted mainly by standardization bodies is so-called *backward compatibility*, i.e., preferring changes that do not influence the existing data. For example, the open XML standard *OpenTravel.org*¹ currently offers 319 XML schemas which standardize communication in the travel industry. OpenTravel.org is changed twice a year and the changes are published in the form of a new version of the XML schemas and a documentation of the changes in a human-readable document. This requires tremendous manual

¹<http://www.opentravel.org>

effort of designers to adapt their transformation scripts and potentially their database, their own XML formats and their program code as well. Thus, OpenTravel.org tries to preserve the backward compatibility as much as possible. The amount of backward-incompatible changes is very small – 7,5% on average within years 2003–2010 (see Figure 1.2). However, this approach results in artificial data structures with plenty of optional items, obsolete data structures, etc. In addition, backward-compatible changes cannot be done in all the cases, so this approach can be used only partly, in specific cases and only until the data structures remain readable and understandable.

Related Problems A natural and real-world solution of the evolution problem is to rely on an IT expert who knows the information system well and is able to denote the part of the system which is modified and

(P1) *to make the required change easily and correctly,*

(P2) *to identify all affected parts of the system*

that need to adapt too, and

(P3) *to make the respective changes of the affected parts semantically correctly.*

But, in a complex information system involving hundreds of schemas it is impossible for a single person to consider and cover all the components and aspects. In addition, since the system may involve multiple formats, the IT expert must know all of them and be able

(P4) *to express the changes also syntactically correctly regarding the selected format.*

And, last but not least, the system may naturally grow, e.g., new schemas may come or be required and, hence, the IT expert must be able

(P5) *to integrate new schemas and discover relations to the current ones.*

Proposed Solution To help to solve the indicated problems P1–P5, in Section 1.1 we describe the idea of a *five-level evolution management framework* we have first proposed in its full generality, i.e., for any kind of data format. Using several levels of abstraction it enables to model all parts of the system regardless technical details of a selected format. Preserved relations between the levels enable to propagate the changes correctly among multiple related and overlapping schemas. In the first part of our research as well as in this thesis we focussed on its first part – so-called *XML view*, i.e., evolution of a set of XML schemas. Hence, in Section 1.1 we also describe the framework from this perspective. In Section 1.2 we focus on the problem of revalidation of XML documents, i.e., instances of evolving XML schemas. We show that having the described levels and preserved relationships among them, the evolution process can be done easily and correctly. Consequently, we solve problems P1–P4.

As we have mentioned, an important question which may arise is how to integrate other XML schemas, i.e., how to deal with the problem of *reverse engineering*. In this case we need to build the system in the bottom-up direction starting from data towards more abstract levels. In Section 1.3 we describe the problem of inference of an XML schema from a set of XML documents and in Section 1.4 we deal with the problem of similarity evaluation which is an important part of the process of mutual mapping of XML schemas. In other words, in these sections we also solve problem P5.

Even though there currently exist several evolution management systems, none of them focusses on the described issues from such a general point of view, with a formal background and covering also other related problems such as the reverse engineering steps. Hence, despite we base our proposal on several verified strategies and technologies, we show that such a system can be designed and implemented as a robust and general tool. Our idea has been first implemented as a project called *XCase* (<http://xcase.codeplex.com/>) and later re-engineered into a system called *eXolutio* (<http://exolutio.com/>). Currently, there exists also a more general implementation of the idea called *DaemonX* (<http://daemonx.codeplex.com/>). All the three systems were created as student SW projects and later improved within Master/Ph.D. theses, all under supervision of members of XRG.

The last but not least separate but related research topic we want to cover in this thesis is the knowledge of real-world data. Such information is important in general; in our case we exploited it for the purpose of proposal of edit operations, change management and similarity matching. In Section 1.5 we

describe a general system called *Analyzer* which eases the analytical process using a set of crawlers, data correctors, analyzers, result visualizations, etc.

1.1 Five-Level Evolution Management Framework

The core of our research forms the five-level evolution management framework described with all technical details in [34] (see Chapter 2). The full architecture of the framework is depicted in Figure 1.3 (see page 7). As we can see, the framework can be partitioned both horizontally and vertically; in both cases its components are closely related and interconnected.

If we consider the vertical partitioning, we can identify multiple views of the system. We have depicted the three most common and representative views. The blue (leftmost) part covers an *XML view* of the data processed and exchanged in the system. The green (middle) part represents the *storage view* of the system, e.g., a relational view of the processed data which need to be persistently stored. Finally, the yellow (rightmost) part represents a *processing view* of the data, e.g., processing by sequences of Web Services described, e.g., using BPEL scripts [5].

If we consider the horizontal partitioning, we can identify five levels, each representing a different view of the system and its evolution. The lowest level, called *extensional level*, represents the particular instances that form the implemented system such as, e.g., XML documents, relational tables or Web Services that are components of particular business processes. Its parent level, called *operational level*, represents operations over the instances, e.g., XML queries over the XML data expressed in XQuery [9] or SQL/XML [3] queries over relations. The level above, called *schema level*, represents schemas that describe the structure of the instances, e.g., XML schemas or SQL/XML Data Definition Language (DDL).

Even these three levels indicate problems related to evolution. For instance, when the structure of an XML schema changes, its instances, i.e., XML documents, and related queries must be adapted accordingly so that their validity and correctness is preserved respectively. In addition, if we want to preserve optimal query evaluation over the stored data, the storage model also needs to adapt respectively. What is more, as we have mentioned, in practice there are usually multiple XML schemas (or schemas in other for-

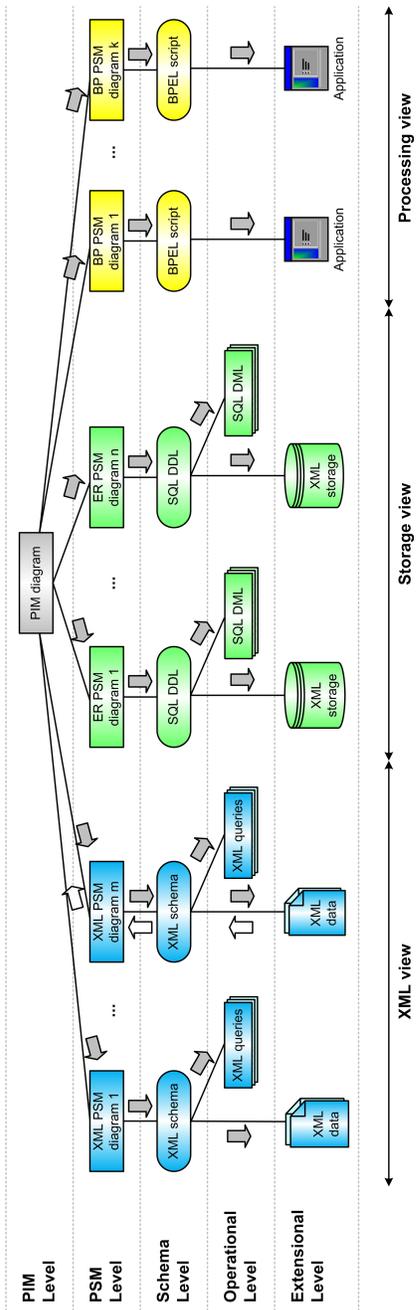


Figure 1.3: Five-level evolution management framework

mats) applied in a single system, i.e., multiple views of the common problem domain. In general, a change at one level can trigger a cascade of changes at other levels.

Considering only the three levels leads to evolution of each affected schema separately. However, this is a highly time-consuming and error-prone solution since we need the IT expert who is able to identify all the affected schemas and propagate the changes, i.e., who knows the mutual relations among the schemas. Therefore, we introduce two additional levels, which follow the *model driven architecture* (MDA) [28] principle, i.e., modeling of a problem domain at different levels of abstraction. The topmost one is the *platform-independent level* which comprises a *schema in a platform-independent model* (*PIM schema*). The PIM schema is a conceptual schema of the problem domain. It is independent of any particular data (e.g., XML or relational) or business process (e.g., Web Services) model. The level below, called *platform-specific level*, represents mappings of the selected parts of the PIM schema to particular data or business process models. For each model it comprises *schemas in a platform-specific model* (*PSM schemas*) such as, e.g., XSEM [33] schemas which model hierarchical data structures implemented using a selected XML schema language or ER [11] schemas which are typically implemented using relational schemas. Each PSM schema can be then automatically translated into a particular language used at the schema level (e.g., XML Schema [42] or SQL DDL [17]) and vice versa.

Now, having a hierarchy of models which interconnect all the applications and views of the data domain using the common PIM level, change propagation can be done semi-automatically and much easily. We do not need to provide a mapping from every PSM to all other PSMs, but only from every PSM to the PIM. Hence, the change propagation is realized using this common point. For instance, if a change occurs in a selected XML document, it is first propagated to the respective XML schema, PSM schema and, finally, PIM schema. We speak about *upwards propagation*, in Figure 1.3 represented by the white arrows. It enables one to identify the part of the problem domain that was affected. Then, we can invoke the *downwards propagation* and propagate the change of the problem domain to all the related parts of the system. In Figure 1.3 it is denoted by the grey arrows.

Related Work Naturally, the idea of evolution management and change propagation is not new. Regarding the related work, we can find a significant

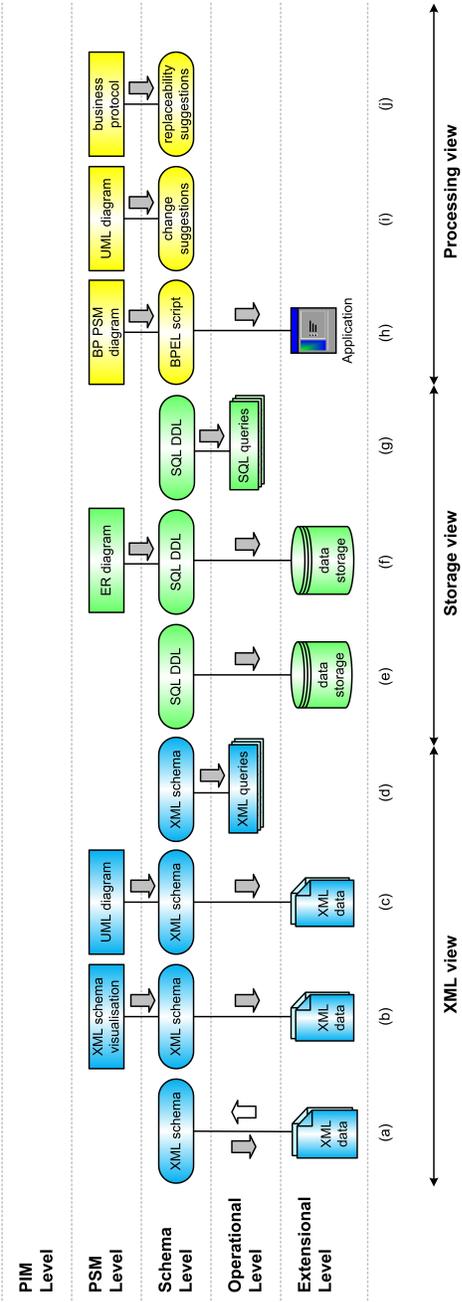


Figure 1.4: Current evolution management approaches

amount of papers that focus on change management and propagation, model transformation, evolution, versioning, etc. The current approaches towards evolution management can be classified according to distinct aspects [15, 25]. The changes and transformations can be expressed [10, 35] as well as divided [14] variously too. However, to our knowledge there exists no general framework comparable to our proposal depicted in Figure 1.3. Particular cases and views of the problem have previously only been solved separately, superficially or mostly imprecisely without any theoretical or formal basis. Their graphical overview is depicted in Figure 1.4 (see page 9), where sections (a)–(j) represent groups of papers that focus on the particular part of the problem in the context of our proposal. A detailed analysis of the related work can be found in paper [34] (see Chapter 2), Section 8.

1.1.1 XML View

As we have mentioned, in the first step of our research as well as in this thesis we focus on the blue part of the system, i.e., the XML view. As depicted in Figure 1.5 (see page 11), even considering only this area, we need to solve several related issues that ensure that the system works correctly.

First of all, we need to define the PIM and PSM levels and the related mapping to neighboring levels, i.e., the *PIM-to-PSM mapping* and the *PSM-to-schema mapping* as depicted in Figure 1.5 (a) with the red color. Regardless the choice of the subproblem, the schema in PIM still models real-world concepts and the relationships between them without any details of their representation in a specific data model. Hence, as a PIM we use the classical model of UML class diagrams [36, 37]. For simplicity, we use only its basic constructs: classes, attributes and binary associations. A schema in PSM describes how a part of the reality modeled by the PIM schema is represented with a particular XML schema. For each aimed XML schema a separate PSM schema is created. As a PSM we use UML class diagrams extended for the purposes of XML modeling. The extension is necessary because of several specifics of XML (such as hierarchical structure or distinction between XML elements and attributes) which cannot be modeled by standard UML constructs. The definition is a modified and for the purpose of clearer explanation of further concepts also simplified version of the *XSEM model* [33] proposed by Martin Načaský. An example of a PIM schema, two PSM schemas and respective XML schemas is depicted in Figure 1.6 (see page 13). The arrows among levels indicate the mapping.

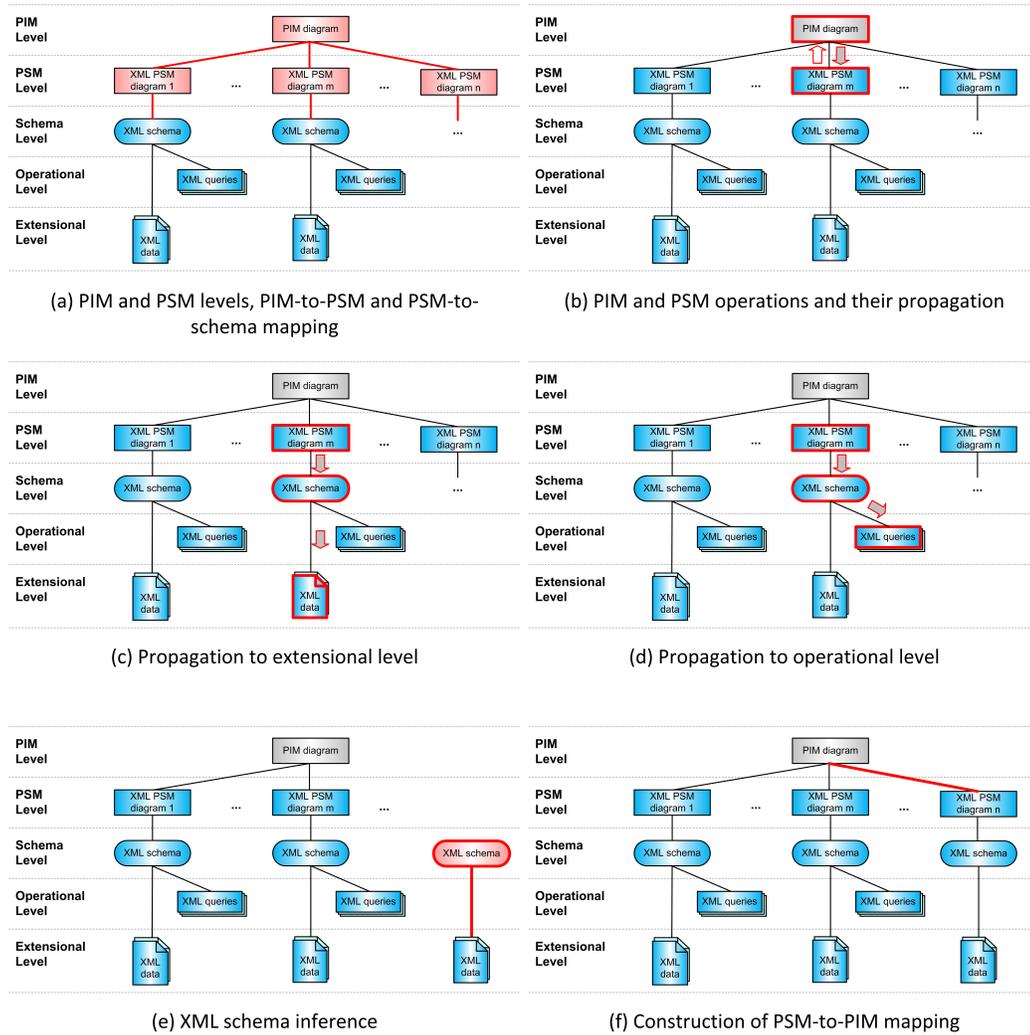


Figure 1.5: Building the XML view of the five-level evolution management framework

Other levels of the XML view, i.e., schema, operational and extensional are defined by the W3C specifications; hence, we can focus on the two conceptual levels (PIM and PSM) and their relations to other levels. From the *conceptual perspective*, a PSM schema is mapped to a PIM schema and models the same part of the reality. From the *grammatical perspective*, a PSM schema models an XML schema, i.e., a regular tree grammar [32]. The mappings are crucial for correct evolution of the XML formats. Whenever a change made to any part of the framework is performed by a user, it is propagated to all other affected parts. The need for change propagation is invoked by the mappings. The propagation ensures that the affected parts are adapted so that their consistency with the initial changed part as well as with each other is preserved.

In the second step, we need to specify the operations at the PIM and PSM levels and their respective propagation (as denoted in Figure 1.5 (b) on page 11 with the red color). As for the operations, we have chosen the approach similar to many other papers. We define a set of *atomic* operations and the way how more complex, so-called *composite* operations can be created from them. To ensure correct propagation mechanism, we also need to define a set of pre- and post-conditions that must be fulfilled with regard to all related components of the framework. The atomic operations at both the PIM and PSM level naturally involve operations *create*, *delete* and *update* for all constructs of the model. However, contrary to the related work, having not only the currently edited model, but also the mapping to other models, we need also other types of operations. That is why we also define operation *synchronization* which specifies that two (sets of) components are semantically equivalent. This operation ensures correct propagation for instance in the simple motivating example (see page 2) when the element `address` should be divided into elements `street`, `city`, `zip`, etc. Simple replacing of the old element with the new ones would cause loss of information (data instances), whereas synchronization ensures that the data are correctly “transformed” between the elements and the transformation is correctly propagated to the rest of the system.

1.2 Adaptation of XML Data

Having described the conceptual levels (PIM and PSM), their mutual relations and the relation to the schema level, the set of respective operations

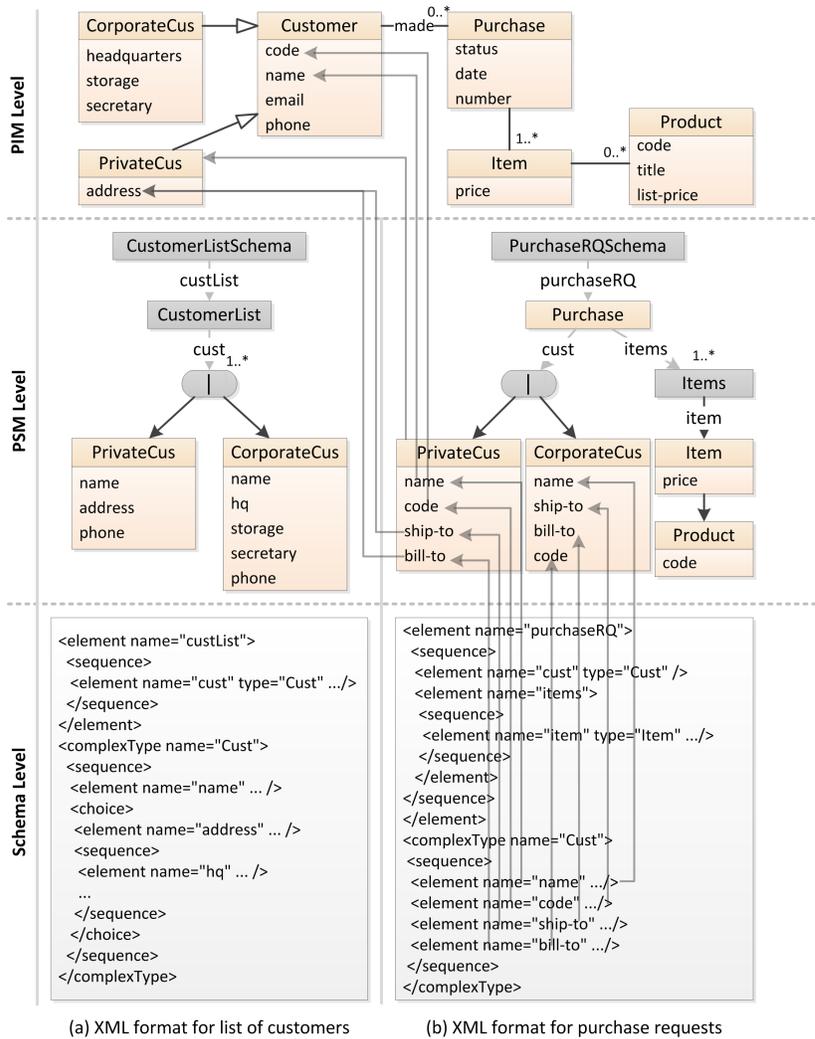


Figure 1.6: Sample PIM schema, two PSM schemas and respective XML schemas

and their propagation, we can now express changes at any of the three levels and be sure that they are correctly propagated to all affected parts at all the three remaining levels.

In paper [24] (see Chapter 3) we incorporate the extensional level and we focus on the problem of adaptation of schema instances, i.e., XML documents, with regard to changes specified at schema level (denoted in Figure 1.5 (c) on page 11 with the red color). Using the previous results, we can consider that the particular XML format represented by the changed schema is changed at the PSM level, i.e., we can abstract from technical details of a particular XML schema language and the user is provided with a more user-friendly tool. However, our aim is more general. We want:

1. to allow the user both to evolve existing schemas and to create new versions schemas, and
2. to let the user work with all versions of the schema.

In other words, each version must be independent of the others and the old version should not be lost and replaced by a new version. The user can choose any existing version v of the whole model and via the *branch* operation, create a new version \tilde{v} as a copy of v . Then, (s)he can evolve \tilde{v} to a desired state, but also go back to work with v or any other version existing in the system.

When designing an adaptation algorithm, it is possible to take one of two distinct approaches: either (1) *recording the changes* as they are conducted during the schema evolution phase or (2) *comparing two versions of the schema*. A system that records changes usually provides a kind of a command that initiates the recording and after issuing this command all operations carried out by a user over the schema are recorded. When the recording is finished, the system can *normalize* the sequence, e.g., by eliminating operations that cancel each other. An alternative approach is to compare the two versions of the schema. The user can work with both schemas independently until (s)he is satisfied with them. The change detection algorithm then takes the schemas as an input and compares them. The result of the comparison is a list of differences between the schemas. In general, schema comparison has many advantages, such as no need to look for redundancies in sequences of operations, simple handling of reverse operations, ability of integration of a schema edited outside the system, etc. But it must solve the problem of

ambiguities. For example, we may not be able to decide whether a particular construct was renamed (and slightly changed), or an old construct was deleted and a new one (with highly similar structure) was created. Or, a similar problem may occur if we consider operation move versus operations add and delete. If we do not want to settle for heuristics, the only correct solution is to find all possible interpretations of a particular change and then let the user select the correct one.

In our approach, we decided to solve this issue by adding another type of concepts into the model – so-called *version links*. Version links connect constructs (i.e., classes, attributes, etc.) that represent the same real-world concept in different versions of the model. They work as a mapping between different versions of the schema. However, the difference between relying upon the user to specify the mapping where ambiguities exists and using version links is crucial. Version links are (most of the time) kept and managed automatically at the background. Each time the user performs the branch operation, version links are created between all the concepts and their new versions. After that, they are maintained until a concept is deleted. Since each construct in the model is correctly connected with its counterpart constructs in all other versions where it exists, we can avoid ambiguities when detecting changes. One can regard version links as an adoption from the change recording approaches and the approach can thus be considered as a combination of schema comparison and change recording.



The approach then outputs an XSLT [19] script that adapts the modified XML documents with regard to the new version of a schema, whereas the adapted document preserves semantical meaning of the constructs with regard to a given PIM.

1.3 Reverse Engineering: Inference of XML Schemas

To ensure correct propagation of changes to all the affected parts of the system, we need to correctly build the whole five-level evolution management framework including the mappings. For this purpose the user can select from two strategies. When the user wants to design a new XML format, (s)he proceeds in the *top-down* direction from the PIM level to the schema level and we therefore call the methodology *forward engineering*. The result of

the process are (eventual extensions to) the PIM schema, the derived PSM schema and the respective XML schema, together with mappings between them. Nevertheless, the user can also integrate existing XML schemas into his/her solution in the *bottom-up* direction which we call *reverse engineering*. Such XML schema might be a legacy XML schema or an XML schema given by a standardization organization we want to use in our system (e.g., a subset of the OpenTravel.org schemas – see the real-world example on page 3).

A quite common situation is when we are provided only with a set of XML documents that represent a particular view of the problem domain, i.e., the extensional level. The documents conform to an XML schema; however, it is not provided. In fact, according to statistical analyses of real-world XML data, a significant portion of real-world XML documents (52% [26] of randomly crawled or 7.4% [31] of semi-automatically collected²) have no schema at all. So, first we need to *infer* the respective XML schema, i.e., a general description of the structure of the data that are supposed to be integrated. In other words we need to build the schema level (as denoted in Figure 1.5 (e) on page 11 with the red color).

The problem of automatic inference of an XML schema for a set of XML documents can be viewed as a problem of construction of a grammar for a set of words. But, since according to Gold’s theorem [16] regular languages (in our case XML schemas) are not *identifiable* from *positive examples* only (in our case sample XML documents which should conform to the resulting schema), the existing methods either infer an identifiable subclass of regular languages or exploit a kind of a heuristic. Since the problem of the former approaches is which subclass should be inferred, most of the current ones rather focus on a heuristic strategy. We cannot specify particular features of the resulting set of languages; however for most real-world applications it is not necessary.

The basic inference process (used in both types of strategies) works as follows: Consider two sample XML documents in Figure 1.7. First, the input documents are transformed to a set of prefix tree automata (see step 1), each for a specific element name. Then, using a set of heuristic rules the states of the automaton are merged to provide a more realistic and general result. An example of such a rule may be that “we can merge states with the same suffix of length $\geq k$ ” (see step 2) or “if there are more than n consequent occurrences of a label e , it is probable that it can occur an arbitrary number

²Data collected with the interference of a human operator.

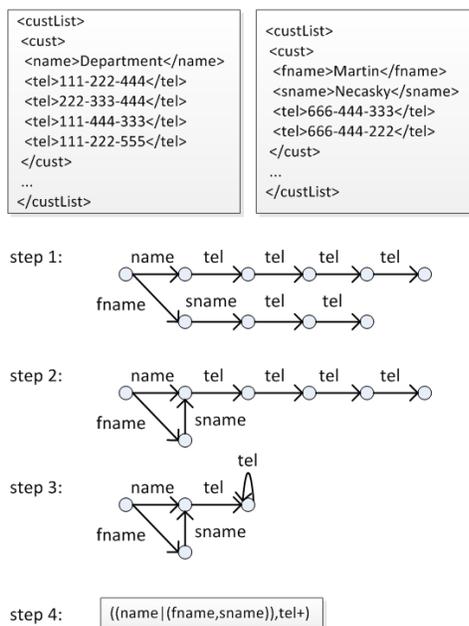


Figure 1.7: An example of the inference process

of times” (see step 3). And, finally, the merged automaton is transformed to an expression in the respective schema language (see step 4, where DTD [7] is used due to space limitations).

For our purpose we also do not need a specific class of languages, but a realistic and precise XML schema that describes the given data. Most of the current approaches [29] focus on inference of DTDs and especially the respective regular expressions. However, our PSM model is richer and thus we have focused on XML Schema definitions (XSDs) [42] and especially on two targets – more precise and realistic results and optimization of the inference process. In paper [30] (see Chapter 4) we provide an overview of existing approaches to XML schema inference and, especially, several improvements of the classical inference process, such as:



- inference of specific XML Schema constructs such as, e.g., unordered sequences, elements with the same name, but different context, or elements with different names but the same content,
- exploitation of user interaction which significantly optimizes the inference process (when used appropriately),

- exploitation of analysis of both structural and semantic similarity of the data to detect more precisely whether a particular part of a schema should be described using inheritance of complex types, shared element/attribute groups or separate schema particles, or
- inference of integrity constraints, in particular XSD keys and foreign keys utilizing an approach originally proposed for relational data.

1.4 Reverse Engineering: Mapping between Schemas

Having a newly coming XML schema, either provided by the user or inferred from sample XML documents, its integration to an existing system is a two-step process. First, we need to acquire its PSM diagram. In this case, the translation between an XML schema and a PSM schema is a straightforward process given by the definition of the model [33]. Second, we need to map the PSM diagram to the existing PIM diagram, i.e., we need to construct the mapping of PSM to PIM (denoted in Figure 1.5 (f) on page 11 with the red color). In this case the situation is much more complex than in case of the top-down direction when the user directly provides the mapping denoting parts of PIM to be interpreted in PSM.

We dealt with the mapping process, e.g., in [22, 40]. The algorithm is based on suggestions of likely candidates for correct interpretation to the user. These suggestions are sorted according to a similarity value computed using well-known similarity techniques combined with our structural similarity technique which exploits the knowledge of correct mappings of parts of the PSM. The strategies result from our older and more general paper [45] (see Chapter 5) which studies various aspects of similarity of XML schemas. We deal with similarity of XML schema fragments expressed in DTD or XML Schema. In particular, we focus on *quantitative* measure which is currently used in a huge number of applications such as clustering of XML data, dissemination-based applications, schema integration systems or other less obvious areas, such as, e.g., e-commerce or semantic and approximate query processing. The problem we are facing is that the key emphasis is currently put on semantic similarity. However, since for our application of similarity evaluation the structure of schema fragments is an important aspect, we focus on more precise structural analysis. On the other hand, since the se-

mantics of the data can also be an important information, we still preserve the exploitation of semantic similarity as well.

To fulfill both the aims, we combine and adapt to DTD constructs two verified approaches – *tree edit distance* and *semantics* of element/attribute names. We show how a well-known and verified methodology of edit distance can be utilized for DTDs, that can involve several types of nodes and form general graphs, and even extended with exploitation of semantic similarity. Using a set of experiments we show the impact of these extensions on similarity evaluation. And, last but not least, we show how this approach can be extended for XSDs, which involve more complex structures than DTDs and, in particular, plenty of “syntactic sugar”, i.e., constructs that are structurally or semantically equivalent. 

1.5 Analysis of Real-World Data

In many aspects related to specification of the five-level evolution management framework, such as schema structures, edit operations, change management or similarity matching, we had to study real-world XML applications, i.e., the data and operations they typically use. However, working with real-world data is not simple, since they can often change, are not precise, or even involve a number of errors. Firstly, we need to gather a reasonably large and representative set of real-world data. Currently there exists a huge number of crawlers, however we usually require data having a particular format or structure, so a range of filters must be supported as well. Secondly, since the data are usually human-written, they contain a number of errors. In this case we can either discard the incorrect data, and, hence, lose a significant portion of them, or provide a kind of *data corrector*. In the next step we want to make the analyses themselves. In this case we have to cope with the fact that the data can change, and, hence, the analytical phase must be repeatable and extensible. And, finally, having obtained the results of the statistics, we need to be able to visualize and analyze the huge amount of information efficiently and mutually compare the results.

Since the analysis of real-world data is an important, but at the same time demanding task, as a “side” research direction we have focussed also on this area. In paper [41] (see Chapter 6) we describe the result of our effort – a general framework called *Analyzer* that aims to cope with all the previously named requirements. It provides all the essential functionality for 

easy management of files to be analyzed, configuration and execution of selected analyses and an advanced graphical user interface (GUI) for browsing generated reports. The key advantage of *Analyzer* is extensibility. *Analyzer* provides a general environment, whereas all analytical computations themselves are defined solely within implementation of *plug-ins*. Hence, we have created a unique and universal tool that can be used in many research areas for various optimization purposes. A screenshot of the tool is provided in Figure 1.8.

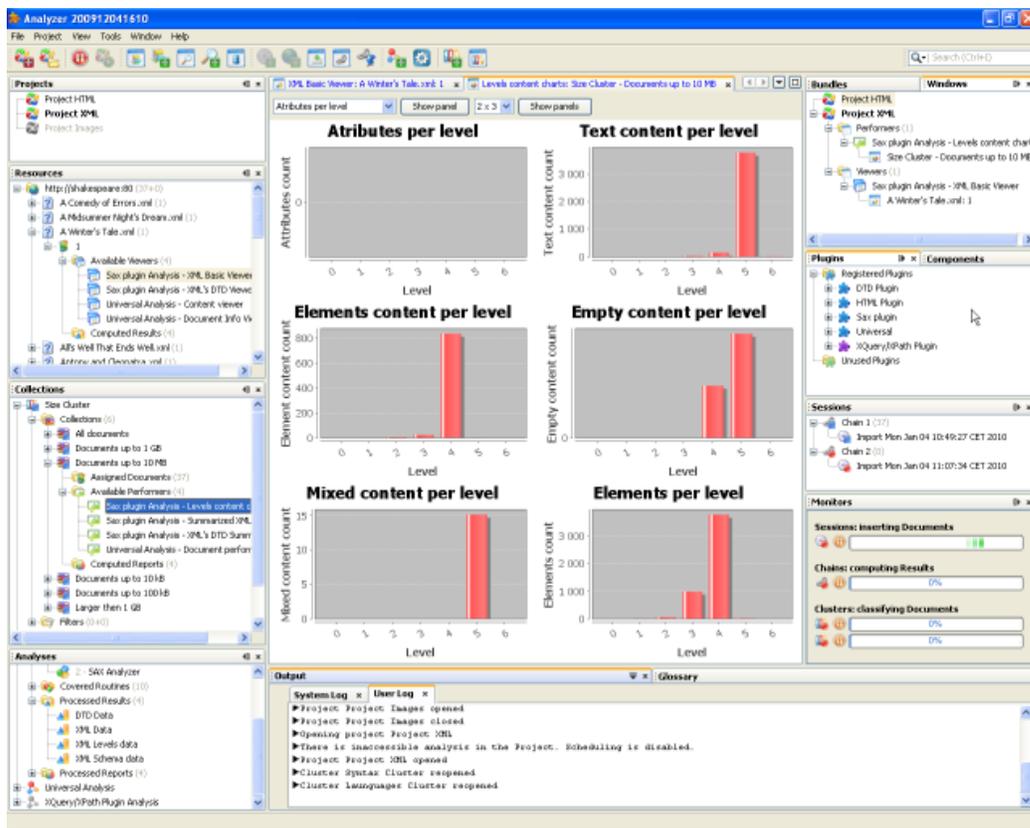


Figure 1.8: A screenshot of *Analyzer*

For our purposes, we have primarily focussed mainly on four related aspects – XML data crawling, XML data correction, structural analysis of XML data, and query analysis of XML queries. In the former three cases we provide significant extensions to the current approaches, in the latter case we provide a unique approach which has not been considered in the



current papers so far. However, despite our original motivations related to XML technologies, we have created a general application that is completely capable of performing analyses over documents of whatever types.

1.6 Author's Contributions

As mentioned before, the five-level evolution management framework is complex, the idea can be further extended and generalized and, hence, the related problems cannot be solved by a single person. On the other hand, a single person cannot deal with all the related issues and problems. The core of the framework is a common work of members of XRG, mainly co-authors of the respective papers that form Chapters 2 and 3. Most parts of the system were also solved within Master theses and student SW projects under supervision of members of XRG, where the contribution of particular students is of different kind, but indisputable and valuable. Similarly, the consecutive Chapters 4, 5, and 6 are a common work of multiple co-authors.

The author of this thesis cooperated on the common core of the five-level evolution management framework, i.e., proposal of the levels, edit operations and propagation of changes within the XML view, on aspects related to reverse engineering, i.e., XML schema inference, XML schema matching and mapping, and on analysis of real-world XML data and operations. Naturally, there are also other related aspects that result from or are related to the same common idea of the five-level evolution management framework, that are beyond the scope of this thesis, or even beyond research topics of the author of this thesis. They are/will be described in Ph.D. and Habilitation theses of the co-authors of the core idea, key contributors of the extensions. The most important topics of this kind involve implementation of the evolution management framework, formal background of the conceptual levels, respective operations, and propagation mechanism, preservation of integrity constraints at all the levels, advanced mapping algorithms, technical details of data analyses involving their correction, analysis of operations, etc. Note that some of these topics are partly covered in papers that form this thesis. However, they do not form the key contributions of its author.

Chapter 2

Evolution and Change Management of XML-Based Systems

Martin Nečaský
Jakub Klímek
Jakub Malý
Irena Mlýnková

Published in the *International Journal of Systems and Software*, volume 85,
issue 3, pages 683–707. Elsevier, February 2012. ISSN 0164-1212.

Impact Factor: 0.836
5-Year Impact Factor: 1.117





Evolution and change management of XML-based systems

Martin Nečaský*, Jakub Klímeček, Jakub Malý, Irena Mlýnková

Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, Malostranské nám. 25, 118 00 Praha 1, Czech Republic

ARTICLE INFO

Article history:

Received 25 February 2011
Received in revised form 20 July 2011
Accepted 18 September 2011
Available online 29 September 2011

Keywords:

XML data modeling
Model driven architecture
XML schema evolution
Propagation of changes

ABSTRACT

XML is de-facto a standard language for data exchange. Structure of XML documents exchanged among different components of a system (e.g. services in a Service-Oriented Architecture) is usually described with XML schemas. It is a common practice that there is not only one but a whole family of XML schemas each applied in a particular logical execution part of the system. In such systems, the design and later maintenance of the XML schemas is not a simple task.

In this paper we aim at a part of this problem – evolution of the family of the XML schemas. A single change in user requirements or surrounding environment of the system may influence more XML schemas in the family. A designer needs to identify the XML schemas affected by a change and ensure that they are evolved coherently with each other to meet the new requirement. Doing this manually is very time consuming and error prone. In this paper we show that much of the manual work can be automated. For this, we introduce a technique based on the principles of Model-Driven Development. A designer is required to make a change only once in a conceptual schema of the problem domain and our technique ensures semi-automatic coherent propagation to all affected XML schemas (and vice versa). We provide a formal model of possible evolution changes and their propagation mechanism. We also evaluate the approach on a real-world evolution scenario.

© 2011 Elsevier Inc. All rights reserved.

1. Introduction

The eXtensible Markup Language (XML) (Bray et al., 2008) is used by many software systems today to represent and exchange data in a form of *XML documents*. One of the crucial parts of such systems are *XML schemas* which describe structure of the XML documents. Usually, a system does not use only a single XML schema, but a set of different XML schemas, each in a particular logical execution part. We can, therefore, speak about a *family of XML schemas*.

Having a system which exploits a family of XML schemas, we face to the problem of *XML schema evolution*. The XML schemas may need to be evolved whenever user requirements or surrounding environment changes. A single change may influence zero or more XML schemas. Without a proper technique, we have to identify the XML schemas affected by the change manually and ensure that they are evolved coherently with each other. When the XML schemas have already been deployed, in the run-time environment there are also XML documents which might become invalid and need to be, therefore, modified appropriately.

In this paper we focus only on a part of the problem described – coherent evolution of XML schemas according to changing

requirements (see our recent work (Malý et al., 2011) where we discuss the other part of the problem – adaptation of underlying XML documents when their XML schemas evolve). We propose a technique based on the Model-Driven Development (MDD) (Miller and Mukerji, 2003) methodology. We consider modeling the XML schemas at two MDD levels – *platform-independent* and *platform-specific*. First, the whole application data domain is modeled independently of the XML schemas in the form of a platform-independent schema. Then, each XML schema in the family is designed in the form of a platform-specific schema which is mapped to the platform-independent schema. It may be then automatically translated to an expression in a selected XML schema language, e.g. XSD (XML Schema Definition) (Thompson et al., 2004) or RELAX NG (Murata, 2002). The mappings of platform-specific schemas to platform-independent schema naturally support evolution management. A change is explicitly expressed as a change to the platform-independent schema or one of the platform-specific schemas. The mappings allow us to propagate the change between platform-independent and platform-specific levels semi-automatically and evolve the whole family of XML schemas coherently.

Contributions. The key contributions of this paper are as follows:

- formal models for designing XML schemas at platform-independent and platform-specific MDD levels and a set of atomic operations for their evolution,

* Corresponding author.

E-mail addresses: necasky@ksi.mff.cuni.cz (M. Nečaský), klimek@ksi.mff.cuni.cz (J. Klímeček), maly@ksi.mff.cuni.cz (J. Malý), mlynkova@ksi.mff.cuni.cz (I. Mlýnková).

- proof of minimality and correctness of the set,
- mechanism for propagating changes invoked by the atomic operations between the MDD levels,
- specification of operations composed of the atomic ones and their propagation between the MDD levels,
- implementation of the proposed framework called *eXolutio* (Klímeček et al., 2011), and
- experimental demonstration of the completeness of the set of atomic operations and correctness of the propagation mechanism by applying *eXolutio* in a real-world case study.

Although there is other existing work in the area of schema evolution (as we will show in Section 8), the evolution problem has not yet been adequately solved (Hartung et al., 2011). We will show that the current approaches omit some important kinds of operations and provide an insufficient solution to the problem of the propagation of changes. And, last but not least, they do not introduce operations as a formal set of simple atomic operations which would allow authors of tools for schema evolution to build various more-user friendly operations as compositions of the atomic operations. In this work, we introduce such formalism. Its main advantage is that it enables one to specify a new operation as a sequence of the atomic operations without the details of how the new operation is propagated to the other parts of the system. Our propagation mechanism ensures its correct propagation automatically.

In this paper we combine and, in particular, extend our previous work in this area. Our technique for designing XML schemas at platform-independent and platform-specific levels was firstly proposed in Nečáský (2009) and later generalized in Nečáský and Mlýnková (2010). A basic implementation of a modeling tool based on the models was introduced in Nečáský et al. (2008). In this text we describe it in detail including its evolution extension and show its usage in real-world use cases. In Nečáský and Mlýnková (2009a,b) we proposed a five-level XML evolution framework which presents a general overview of the problem of XML schema evolution in the context of a whole software system consisting of various parts. In this paper we lay the theoretical basis of our approach. We provide a formal and detailed description of evolution operations and their propagation, prove minimality and the correctness of our approach and extend it with explanatory examples. In general, this paper expands on the results of our recent research with an emphasis on formal specification.

In Yu and Popa (2005) the authors discussed two kinds of evolution approaches – *incremental* and *change-based approaches*. An incremental approach enables a clear formal basis which ensures correctness and allows for simple evolutionary steps made by a

designer. A change-based approach is suitable for cases when we are provided with two versions of the schema without the incremental evolutionary steps and we need to manage evolution of the data efficiently. In this work, we introduce an incremental approach based on a set of atomic operations. A designer incrementally performs particular atomic operations or operations comprising the atomic ones. Our technique continuously propagates the changes to affected schemas.

Outline. The rest of the paper is structured as follows: In Section 2 we provide a motivating and running example. In Section 3 we describe the problem of XML schema evolution in the context of a whole software system and specify the selected part of the problem solved in this paper. In Section 4 we provide a formal specification of platform-independent and platform-specific levels for XML schema modeling. In Section 5 we extend the levels with a set of atomic operations. In Section 6 we describe the propagation mechanism of the atomic operations between the levels and show that the atomic operations together with the propagation mechanism form a minimal and correct evolution formalism. In Section 7 we show how the atomic operations form realistic composite operations. In Section 8 we compare our proposal with current related works. In Section 9 we introduce the implementation of the introduced evolution formalism called *eXolutio* and its application in a real-world case study. We also evaluate our approach on the basis of this case study. Finally, in Section 10 we conclude and outline possible future work.

2. Motivating and running example

As a demonstration of the problem of evolution management of XML schemas, let us consider a company that receives purchase orders and let us focus on a part of the system that processes purchases. Let the messages used in the system be XML messages formatted according to a family of different XML schemas. Consider the two sample XML documents in Fig. 1. The former one is formatted according to an XML schema specifying a list of customers. The latter one is formatted according to a different XML schema specifying purchase requests. There are also other XML schemas in the family (e.g. customer details, purchase responses, purchase transport details, etc.). All the XML schemas share the same data domain (purchasing goods). On the other hand, the same part of the domain may be represented in different XML schemas in different ways. For example, the concept of *customer* is represented in each of our sample XML schemas in a different way. On the right hand side, elements `name` and `email` are present for a customer. On the left hand side, kinds of customers are distinguished (private

```
<custList version="1.3">
  <cust>
    <name>Martin Necasky</name>
    <address>Vaclavske nam. 123, Prague</address>
    <phone>123 456 789</phone>
  </cust>
  <cust>
    <name>Department of Software Engineering,
      Charles University</name>
    <hq>Malostranske nam. 25, Prague</hq>
    <storage>Ke Karlovu 3, Prague</storage>
    <secretary>Ke Karlovu 5, Prague</secretary>
    <phone>111 222 333</phone>
  </cust>
</custList>
```

```
<purchaseRQ version="1.0">
  <bill-to>Malostranske nam. 25, Prague</bill-to>
  <ship-to>Ke Karlovu 3, Prague</ship-to>
  <cust>
    <name>Department of Software Engineering,
      Charles University</name>
    <email>ksi@mff.cuni.cz</email>
  </cust>
  <items>
    <item>
      <code>P045</code>
    </item>
    <item>
      <code>P332</code>
    </item>
  </items>
</purchaseRQ>
```

Fig. 1. Sample XML documents represented in a single XML system.

and corporate customers). For private customers, elements `name`, `address` and `phone` are present. For corporate customers, elements `name`, `different addresses` (`headquarters`, `storage` and `secretary`), and `phone` are present.

Let us consider a new user requirement that an address should no longer be represented as a simple string. Instead, it should be divided into elements `street`, `city`, `zip`, etc. Such a situation would require a skilled domain expert to identify all the schemas in the system which involve an address and correct them respectively. Apparently, in a complex system comprising tens or even hundreds of schemas, this is a difficult and error-prone task. Even identifying the affected parts of the schema is not an easy and straightforward process. For example, we may need to make the modification only for addresses that represent a place to ship the goods (which are the elements `address` and `storage` in the XML schema instantiated on the left-hand side of the figure and element `ship-to` on the right-hand side). We do not want to modify addresses that represent headquarters, etc. In the following text we show in detail that evolution management is a complex process that can be solved semi-automatically and, hence, efficiently and precisely if we provide a rigorous theoretical background and preserve nontrivial relations and meta-data.

3. XML evolution framework

In our previous work (Nečaský and Mlýnková, 2009a), we introduced a framework for managing evolution of a software system which exploits XML technologies at different levels. An extended version of the framework is depicted in Fig. 2. As we can see, the framework can be partitioned both horizontally and vertically; in both cases its components are closely related and interconnected. The relations form the key concept of the evolution management, since they invoke the needs for change propagation.

If we consider the vertical partitioning, we can identify multiple views of the system. In the framework we have depicted the three most common and representative views. The blue (leftmost) part covers an XML view of the data processed and exchanged in the system. The green (middle) part represents the storage view of the system, e.g. a relational view of the processed data which need to be persistently stored. Finally, the yellow (rightmost) part represents a processing view of the data, e.g. processing by sequences of Web Services described using BPEL scripts (WSBPEL, 2007) or various proprietary formats (e.g. Park and Park, 2008).

If we consider the horizontal partitioning, we can identify five levels, each representing a different view of an XML system and its evolution. The lowest level, called *extensional level*, represents the particular instances that form the implemented system such as, e.g., XML documents, relational tables or Web Services that are

components of particular business processes. Its parent level, called *operational level*, represents operations over the instances, e.g. XML queries over the XML data expressed in XQuery (Boag et al., 2007) or SQL/XML (ISO/IEC, 2006) queries over relations. The level above, called *schema level*, represents schemas that describe the structure of the instances, e.g. XML schemas or SQL/XML Data Definition Language (DDL).

Even these three levels indicate problems related to XML evolution. For instance, when the structure of an XML schema changes, its instances, i.e. XML documents, and related queries must be adapted accordingly so that their validity and correctness is preserved respectively. In addition, if we want to preserve optimal query evaluation over the stored data, the storage model also needs to adapt respectively. What is more, as we have mentioned, in practice there are usually multiple XML schemas (families of XML schemas) applied in a single system, e.g. XML schemas for purchases, invoices, product catalogues, etc., i.e. multiple views of the common problem domain. Hence, such a change can influence multiple XML schemas, XML documents and queries. In general, a change at one level can trigger a cascade of changes at other levels. We call such sequences of adaptations *change propagation*.

Considering only the three levels leads to evolution of each affected schema separately. However, this is a highly time-consuming and error-prone solution since we need a domain expert who is able to identify all the affected schemas and propagate the changes. Therefore, we introduce two additional levels, which follow the MDD (Miller and Mukerji, 2003) principle, i.e. modeling of a problem domain at different levels of abstraction. As we have mentioned, the topmost one is the *platform-independent level* which comprises a *schema in a platform-independent model (PIM schema)*. The PIM schema is a conceptual schema of the problem domain. It is independent of any particular data (e.g. XML or relational) or business process (e.g. Web Services) model. The level below, called *platform-specific level*, represents mappings of the selected parts of the PIM schema to particular data or business process models. For each model it comprises *schemas in a platform-specific model (PSM schemas)* such as, e.g., XSEM schemas (Nečaský, 2009) which model XML schemas, ER (Chen, 2002) schemas which model relational schemas, etc. Each PSM schema can be then automatically translated to a particular language used at the schema level and vice versa. Note that the latter direction allows for integration of incoming formats/applications into the given evolution framework.

As we can see in Fig. 2, there are not only vertical relations between the levels, but the components of the system can also be horizontally related across the vertical partitions. A few examples are denoted by the red dashed arrows. For instance, there is a relation between an XML schema and its respective storage in a relational database. Similarly, an XML query can be evaluated by

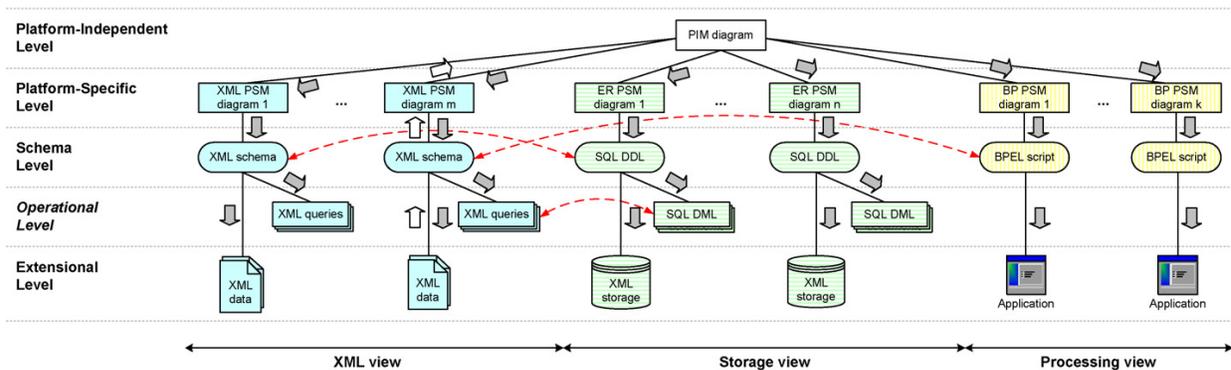


Fig. 2. Five-level XML evolution architecture.

translation into an SQL query. And, last but not least, a BPEL script can specify how an input SOAP message, i.e. an XML schema, is processed.

In all the three cases a change in one of the ends of the relation influences the other. So, since we are considering completely different formats involving different constructs which do not have to correspond mutually using one-to-one relationship, the change propagation becomes a complex problem. But, having a hierarchy of models which interconnect all the applications and views of the data domain using the common PIM level, it can be done semi-automatically and much more easily. We do not need to provide a mapping from every PSM to all other PSMs, but only from every PSM to the PIM which is, in addition, quite natural. Hence, the vertical change propagation is realized using this common point. For instance, if a change occurs in a selected XML document, it is first propagated to the respective XML schema, PSM and, finally, PIM. We speak about an *upwards propagation*, in Fig. 2 represented by white arrows. It enables one to identify the part of the problem domain that was affected. Then, we can invoke the *downwards propagation*. It enables one to propagate the change of the problem domain to all the related parts of the system. In Fig. 2 it is denoted by grey arrows.

3.1. Selected part of the problem

Apparently, the change propagation problem is not an easy task and cannot be covered in a single paper. In this paper we aim at one particular problem – XML schema evolution. As we have shown in the motivating example, there is usually a whole family of XML schemas which are conceptually related to the problem domain of the system. When a designer needs to make a change in one of the XML schemas, the other XML schemas may be affected as well. We introduce an approach which is based on modeling the changes at PIM and PSM XML levels as highlighted in Fig. 3. It ensures that whenever a change is performed in a PIM schema, it is correctly propagated to the PSM schemas and vice versa. So it ensures consistency between the schemas when they are changed.

In practice, this problem appears in two scenarios. The first scenario is when a designer creates *new XML schemas* which have not been deployed in a run-time environment yet. There are neither XML documents formatted according to the XML schemas, nor other developers or applications which would somehow use the XML schemas. In other words, there is no extensional level and no operational level. Because of the complexity of the task, the designer does not create XML schemas in a single linear process. Instead, (s)he iterates in several cycles before an acceptable version of the XML schemas is prepared to be deployed. (S)he starts each iteration with a selected part of the requirements and incorporates

them into the XML schemas. For that, (s)he needs a mechanism which shows an impact of a next change to the unfinished XML schemas and which helps to adapt the XML schemas according to this change. No propagation to extensional or operational levels is necessary at this stage.

The most frequent modifications to the XML schemas in this scenario will be, intuitively, creating new parts of the XML schemas. However, updating existing parts with their more detailed and elaborate variants will be frequent as well. This is because the designer will cover some of the requirements only briefly in the XML schemas in early iterations and will return to them in later iterations to finish them. In simpler cases, updating means changing properties of existing XML schema components (e.g. data type). In more complex cases, updating means removing old parts and replacing them with new but semantically equivalent and more elaborate parts. No backward compatibility of the new version needs to be preserved since there is neither extensional, nor operational level.

The second scenario is *adapting existing XML schemas* which have already been deployed in a run-time environment. In this scenario it is necessary to consider the extensional and operational level as well, because there exist XML documents and applications which use the XML schemas. Such scenario usually occurs when new or changed requirements need to be implemented in the system (e.g. a legislative change). Due to backward compatibility the designer will probably not remove the existing parts of the XML schemas. If some part needs to be detailed (or, conversely, simplified), it will be extended with a new version, not replaced.

The approach we introduce in the following sections is fully sufficient for the first scenario and partly also for the second scenario. For the second scenario, propagation to the extensional and operational level is also necessary. We described the propagation to the extensional level in Malý et al. (2011). The technique introduced generates an XSLT script which transforms XML documents from the old version of each affected XML schema to the new version. The propagation to the operational level is the matter of our future work.

A careful reader might notice that we omitted the schema level in the above paragraphs. Our approach allows the designer to work only at the PIM and PSM levels and not to consider the schema level. This is because our introduced PSM level is equivalent to the schema level from the syntactical point of view. The PSM level has two purposes in addition to the schema level – it provides a more user-friendly presentation of the XML schemas to the designer and extends the XML schemas with mappings to the conceptual schema at the PIM level. In Nečaský and Mlýnková (2010), we proved the equivalence formally. We also showed how a PSM schema may be automatically translated to an XML schema expressed in some XML

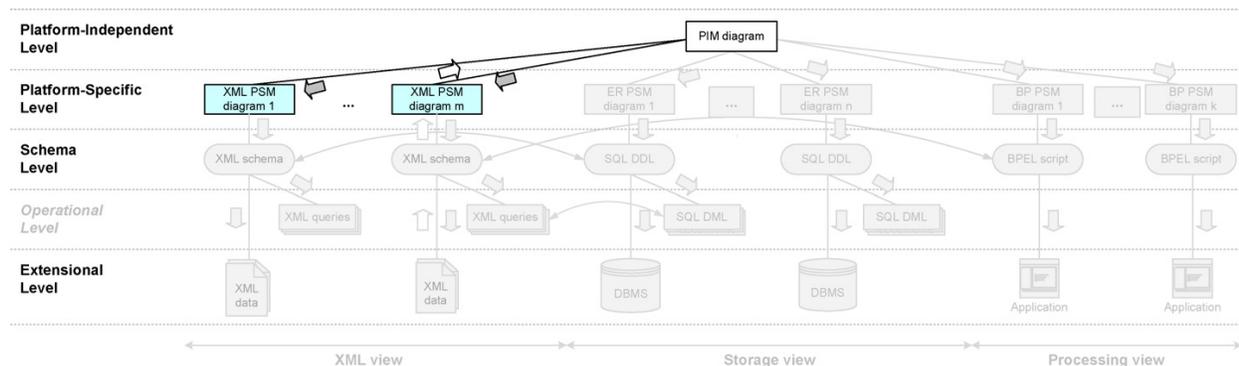


Fig. 3. Five-level XML evolution architecture – data representation.

schema language and vice versa via the formalism of *regular tree grammars*. However, this is beyond the scope of this paper. We will just keep this fact in mind. We exploit this theoretical result in Malý et al. (2011), where we do not generate XSLT scripts on the basis of differences between XML schemas but PSM schemas. We will similarly use it in this paper as well. We will present a set of operations for changing PSM schemas. Because of the equivalence, a change operation at the PSM level unambiguously and correspondingly describes a change in the modeled XML schema and it is not, therefore, necessary to explicitly convert it to a change specific for an XML schema expressed in some XML schema language.

4. Modeling for XML evolution

As we have outlined, our framework enables one to manage evolution of a family of XML schemas by introducing platform-independent and platform-specific levels. In this section, we introduce both levels formally.

4.1. Platform-independent model

A schema in the platform-independent model (PIM) models real-world concepts and the relationships between them without any details of their representation in a specific data model (XML in our case). As a PIM, we use the classical model of UML class diagrams (Object Management Group, 2007a,b). For simplicity, we use only its basic constructs: classes, attributes and binary associations. UML is widely supported by the majority of tools for data engineering and the XMI (XMI, 2009) standard is used for exchanging diagrams between them; it is, therefore, natural to use UML in our approach as well.

Definition 4.1. A *platform-independent schema (PIM schema)* is a triple $S = (S_c, S_a, S_r)$ of disjoint sets of *classes*, *attributes*, and *associations*, respectively.

- Class $C \in S_c$ has a name assigned by function *name*.
- Attribute $A \in S_a$ has a name, data type and cardinality assigned by functions *name*, *type*, and *card*, respectively. Moreover, A is associated with a class from S_c by function *class*.
- Association $R \in S_r$ is a set $R = \{E_1, E_2\}$, where E_1 and E_2 are called *association ends* of R . R has a name assigned by function *name*. Both E_1 and E_2 have a cardinality assigned by function *card* and are associated with a class from S_c by function *participant*. We will call *participant*(E_1) and *participant*(E_2) *participants* of R . *name*(R) may be undefined, denoted by *name*(R) = λ .

For a class $C \in S_c$, we will use *attributes*(C) to denote the set of all attributes of C , i.e. *attributes*(C) = $\{A \in S_a : \text{class}(A) = C\}$. Similarly, *associations*(C) will denote the set of all associations with C as a participant, i.e. *associations*(C) = $\{R \in S_r : (\exists E \in R)(\text{participant}(E) = C)\}$.

PIM schema components have usual semantics: a class models a real-world concept, an attribute of that class models a property of the concept, and, an association models a kind of relationships between two concepts modeled by the connected classes. A sample PIM schema modeling our sample domain of products being sold is depicted in Fig. 4. We display PIM schemas as UML class diagrams. We omit displaying data types of class attributes. When a cardinality of a class attribute or association endpoint is not displayed, it is 1..1 by default.

4.2. Platform-specific model

A schema in the platform-specific model (PSM) describes how a part of the reality modeled by the PIM schema is represented with a particular XML schema. For each aimed XML schema a separate PSM schema is created. As a PSM we use UML class diagrams extended for the purposes of XML modeling. The extension is necessary because of several specifics of XML (such as hierarchical structure or distinction between XML elements and attributes) which cannot be modeled by standard UML constructs.

Definition 4.2. A *platform-specific schema (PSM schema)* is a 5-tuple $S' = (S'_c, S'_a, S'_r, S'_m, C'_{S'})$ of disjoint sets of *classes*, *attributes*, *associations*, and *content models*, respectively, and one specific class $C'_{S'} \in S'_c$ called *schema class*.

- Class $C' \in S'_c$ has a name assigned by function *name*.
- Attribute $A' \in S'_a$ has a name, data type, cardinality and XML form assigned by functions *name*, *type*, *card* and *xform*, respectively. *xform*(A') $\in \{e, a\}$. Moreover, it is associated with a class from S'_c by function *class* and has a position assigned by function *position* within the all attributes associated with *class*(A').
- Association $R' \in S'_r$ is a pair $R' = \{E'_1, E'_2\}$, where E'_1 and E'_2 are called *association ends* of R' . Both E'_1 and E'_2 have a cardinality assigned by function *card* and each is associated with a class from S'_c or content model from S'_m assigned by function *participant*, respectively. We will call *participant*(E'_1) and *participant*(E'_2) *parent* and *child* and will denote them by *parent*(R') and *child*(R'), respectively. Moreover, R' has a name assigned by function *name* and has a position assigned by function *position* within the all associations with the same *parent*(R'). *name*(R') may be undefined, denoted by *name*(R') = λ .

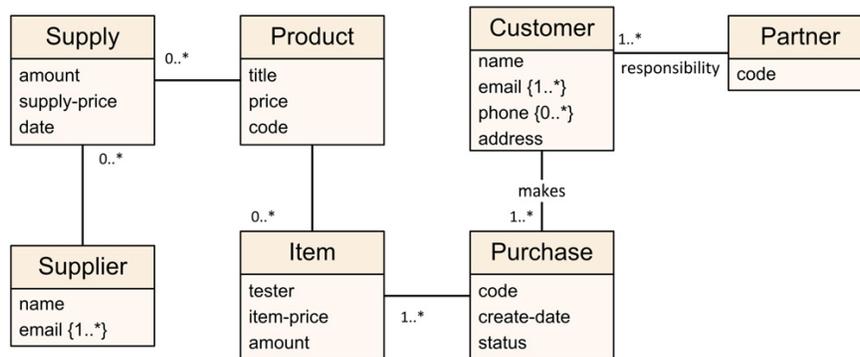


Fig. 4. PIM schema modeling the domain of selling products.

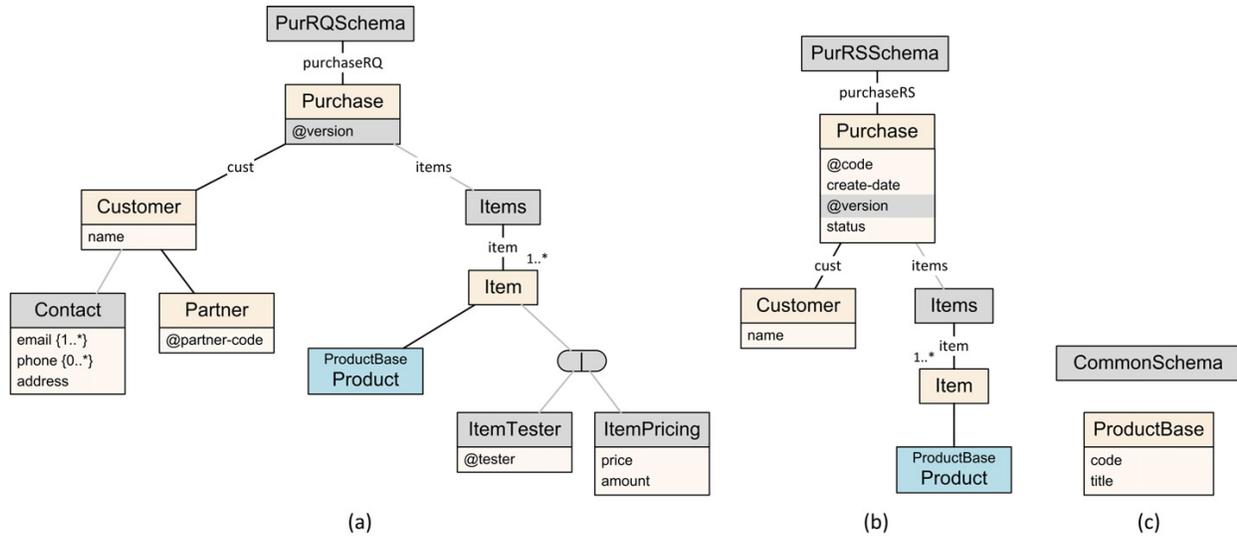


Fig. 5. PSM schema modeling (a) XML format for purchase requests received from customers, (b) XML format for purchase responses sent to customers, (c) components shared by other PSM schemas.

Table 1

XML attributes and XML elements modeled by PSM constructs.

PSM construct	Modeled XML construct
$C' \in S'_c$	Complex content which is a sequence of XML attributes and XML elements modeled by attributes in $attributes(C')$ followed by XML attributes and XML elements modeled by associations in $content(C')$
$A' \in S'_a$, where $xform(A') = a$	XML attribute with name $name(A')$, data type $type(A')$ and cardinality $card(A')$
$A' \in S'_a$, where $xform(A') = e$	XML element with name $name(A')$, simple content with data type $type(A')$ and cardinality $card(A')$
$R' \in S'_r$, where $name(R') \neq \lambda$	XML element with name $name(R')$, complex content modeled by $child(R')$ and cardinality $card(R')$. If $R' \in content(C'_s)$, R' models a root XML element
$R' \in S'_r$, where $name(R') = \lambda$	Complex content modeled by $child(R')$
$M' \in S'_m$ and $cmttype(M') =$ sequence (or choice or set)	Complex content which is a sequence (or choice or set, respectively) of XML attributes and XML elements modeled by associations in $content(C')$

- Content model $M' \in S'_m$ has a content model type assigned by function $cmttype$. $cmttype(M') \in \{\text{sequence, choice, set}\}$.

The graph $(S'_c \cup S'_m, S'_r)$ must be a forest¹ of rooted trees with one of its trees rooted in C'_s . For $C' \in S'_c$, $attributes(C')$ will denote the sequence of all attributes of C' ordered by position, i.e. $attributes(C') = (A'_i \in S'_a : class(A'_i) = C' \wedge i = position(A'_i))$. Similarly, $content(C')$ will denote the sequence of all associations with C' as a parent ordered by position, i.e. $content(C') = (R'_i \in S'_r : parent(R'_i) = C' \wedge i = position(R'_i))$. We will call $content(C')$ content of C' . With $anc(X')$ we will denote the set of all ancestor classes of a component X' in S' .

To distinguish PIM components from PSM components, we strictly use a notation without the ' symbol for PIM components (e.g. class Purchase) and notation with the ' symbol for PSM components (e.g. class Purchase'). Before showing sample PSM schemas, we explain the semantics of the PSM constructs. We view a PSM schema S' from two perspectives: *grammatical* and *conceptual*. From each perspective, the constructs have a different semantics.

¹ Note that since S' is a forest, we could model R' directly as a pair of connected components. However, we use association ends to unify the formalism of PSM with the formalism of PIM.

From the *conceptual perspective*, S' is mapped to a PIM schema S and models the same part of the reality as S . More precisely, some classes, attributes and associations of S' are mapped to some classes, attributes, and associations of S , respectively. These mapped components of S' model exactly the same part of the reality as do their corresponding counterparts in S . The rest of S' has no semantics from the conceptual perspective.

From the *grammatical perspective*, S' models an XML schema. Its components model XML attributes and XML elements, and their structure. We summarize XML constructs modeled by PSM constructs in Table 1. Formally, S' unambiguously models a regular tree language which can be specified by a regular grammar (Murata et al., 2005). However, this formalism is not a part of this paper. For the details on the modeled regular tree language and formal proofs of unambiguity we refer to our previous work (Nečaský and Mlýnková, 2010), where we proved that our PSM is equivalent to regular tree grammars. In other words, it can be equivalently used as an XML schema language. We showed how a PSM schema can be unambiguously translated to an expression in a selected XML schema language and vice versa. The important consequence of our previous results for this paper is that we can abstract our evolution mechanism from particular XML schema languages and work only at the PSM level.

If we put both perspectives together, the PSM schema S' specifies how the corresponding part of the PIM schema S is represented in the XML schema. In other words, it specifies how a part of the

```

<purchaseRQ version="1.0">
  <cust partner-code="PA1">
    <name>Martin Necasky</name>
    <email>necasky@...</email><address>Malostranske nam. 25, Praha, Czech Republic</address>
  </cust>
  <items>
    <item tester="true"><code>P001</code><title>Sample for testing</title></item>
    <item><code>P002</code><title>Umbrella</title><price>100</price><amount>2</amount></item>
  </items>
</purchaseRQ>

```

Fig. 6. Sample purchase request represented in the XML format modeled by the PSM schema depicted in Fig. 5(a).

real world modeled by \mathcal{S} is represented in XML documents valid against the XML schema. Conversely, it specifies the semantics of the XML schema in terms of \mathcal{S} , i.e. the semantics of a particular XML document in terms of the PIM schema.

Three sample PSM schemas are depicted in Fig. 5. We display PSM schemas as UML class diagrams with some extended notation. First, they are displayed in a tree layout; attributes and associations are sorted in the order given by *position*. Second, attributes with XML form a are displayed with the @ symbol. Third, sequence, choice and set content models are displayed as rounded boxes with an inner symbol \dots , | or $\{\}$, respectively.

From the conceptual perspective, our sample PSM schemas are mapped to a part of the PIM schema in Fig. 4. We display the components mapped to the PIM schema in the sea shell color. The mapping is intuitive² and we do not display it explicitly. The components which are not mapped are displayed in grey. For example, the PSM class `Purchase'` in Fig. 5(a) is mapped to the PIM class `Purchase`. In other words, the semantics of `Purchase'` is the same as the semantics of `Purchase` which models purchases. Similarly, attribute `name'` of class `Customer'` is mapped to `name` of class `Customer`. Association `cust'` connecting classes `Purchase'` and `Customer'` is mapped to association `makes` connecting classes `Purchase` and `Customer`. On the other hand, PSM class `Contact'` is not mapped to the PIM schema. In other words, it has no semantics from the conceptual point of view. Similarly, the association with `Contact'` as child is not mapped. And, attribute `version'` of `Purchase'` is not mapped as well. These non-mapped components have no semantic meaning from the conceptual point of view.

From the grammatical perspective, the PSM schema depicted in Fig. 5(a) models an XML schema for purchase requests sent by customers to our system. A sample XML document formatted according to this XML schema is depicted in Fig. 6. The hierarchical structure of the XML schema is modeled by the associations of the PSM schema. As can be seen from the example, each named association models an XML element whose cardinality is given by the child cardinality of the association. For example, association `items'` which connects classes `Purchase'` and `Items'` models XML element `items` with cardinality $1..1$. Association `item'` which connects classes `Items'` and `Item'` models XML element `item` with cardinality $1..*$. Moreover, when such association is in the content of the schema class, it models root XML elements. In our case, association `purchaseRQ'` models XML elements `purchaseRQ` which are root XML elements of the modeled XML format. An association without a name models only the nesting of XML content. For example, association `ItemProduct'` which connects classes `Item'` and `Product'` does not model any XML element. It specifies that the XML content modeled by its child is a part of the XML content modeled by its parent. An attribute models an XML element or XML attribute depending on its XML form. An attribute

with XML form $= a$ models XML attribute and is depicted by the additional symbol @. An attribute with XML form $= e$ models XML element and is depicted without any additional symbol. Again, cardinality is given by the attribute cardinality. For example, attribute `version'` of class `Purchase'` models a mandatory XML attribute `version`. Attribute `name'` of class `Customer'` models a mandatory XML element `name` which can be repeated.

Sometimes, classes in one or more PSM schemas may share the same attributes and/or part of their content. Instead of repeating them at several places, we introduce *structural representatives* which allow for attribute and content reuse. If a class C' in a PSM schema is a structural representative of another class D' from the same or another PSM schema, C' "inherits" the attributes and content of D' . From the grammatical perspective, C' models the same XML attributes as D' followed by its own modeled XML attributes followed by XML elements modeled by D' and, finally, followed by its own modeled XML elements.

Definition 4.3. Let $S' = (\mathcal{S}'_c, \mathcal{S}'_a, \mathcal{S}'_r, \mathcal{S}'_m, \mathcal{C}'_{S'})$ be a PSM schema and C' be a class from \mathcal{S}'_c . C' may be a *structural representative* of another class D' in \mathcal{S}'_c which is assigned to C' by function $repr(repr(C')=D')$. If $repr(C')$ is undefined, denoted by $repr(C')=\lambda$, we say that C' is not a structural representative of any class. Let $repr^*(\lambda)=\{\}$ and $repr^*(C')=\{repr(C')\} \cup repr^*(repr(C'))$ where $C' \neq \lambda$. It must hold that $C' \neq repr^*(C')$.

A structural representative C' of $repr(C')$ is displayed as a class with a blue background and the name of $repr(C')$ above its own name. For example, class `Product'` from Fig. 5(a) and class `Product'` from Fig. 5(b) are both structural representatives of class `ProductBase` from the PSM schema depicted in Fig. 5(c). From the grammatical perspective they both model the same XML fragment as the latter one. Note that the PSM schema in Fig. 5(c) does not model any XML documents (because it does not have any named association going from the schema class and, therefore, does not model any root XML elements). It acts as an auxiliary PSM schema which contains components shared by other PSM schemas via the mechanism of structural representatives.

In the rest of this section we further formalize the conceptual perspective. A formal model of the grammatical perspective is provided in Nečaský and Mlýnková (2010) and we omit it in this paper.

4.3. Formal model of conceptual perspective

Formally, the conceptual perspective of a PSM schema is expressed as a mapping of the PSM schema to the PIM schema. Before we introduce the mapping, we introduce an auxiliary notion of a directed image of an association from a PIM schema which we use in the following definitions.

Definition 4.4. Let $R = \{E_1, E_2\}$ be an association in a PSM schema S . The *directed images* of R are $R^{E_1} = (E_1, E_2)$ and $R^{E_2} = (E_2, E_1)$. We will denote the set of all directed images of S as \tilde{S}_r , i.e. $\tilde{S}_r = \{R^{E_1}, R^{E_2} : R = \{E_1, E_2\} \in S_r\}$.

² The reader may deduce it from their names which intuitively suggest the mapping.

Now, we are ready to introduce the formalism of mappings. We call the mapping of the PSM schema to the PIM schema *interpretation of the PSM schema against the PIM schema*.

Definition 4.5. An *interpretation* of a PSM schema S' against a PIM schema S is a partial function $I : (S'_c \cup S'_a \cup S'_r) \rightarrow (S_c \cup S_a \cup S_r)$ which maps a class, attribute or association from S' to a class, attribute or directed image of an association from S , respectively. For $X' \in (S'_c \cup S'_a \cup S'_r)$, we call $I(X')$ *interpretation of X'* . $I(X') = \lambda$ denotes that X' does not have an interpretation. In that case we will also say that X' has an empty interpretation.

An arbitrary interpretation of a PSM component would lead to inconsistencies between the semantics of the PIM schema and the semantics of the PSM schema given by the interpretation. This would result in ambiguities in the semantics of PSM schemas. For example, suppose the class `Product'` and its attribute `code'` from our sample PSM schema depicted in Fig. 5(a). Let the interpretation of `Product'` be the PIM class `Product`. Therefore, `code'`, from the conceptual perspective, belongs to `Product`. On the other hand, suppose that `code'` is mapped to the PIM attribute `code` of PIM class `Purchase`. From this, `code'` belongs to `Purchase` which is in contradiction with the previous conclusion. We, therefore, need the interpretation to meet certain rules which prevent these ambiguities.

Before we introduce the rules, let us define the notion of *interpreted context* of a PSM component.

Definition 4.6. Let X' be a component of a PSM schema S' . Let I be an interpretation of S' against a PIM schema S . The *interpreted context* of X' with respect to I is denoted $intcontext(X')$ and

- $intcontext(X') = X'$ when $X' \in S'_c$ and $I(X') \neq \lambda$
- $intcontext(X') = C'$ when $X' \notin S'_c$ or $I(X') = \lambda$, where C' is the closest ancestor class to X' s.t. $I(C') \neq \lambda$.

As the definition shows, the interpreted context of each PSM component X' is X' itself if it is a class with an interpretation. In other cases, it is the closest ancestor class to X' . Let us demonstrate the notion of interpreted context on our sample PSM schema depicted in Fig. 5(a). The interpreted context of class `Customer'` is class `Customer'` itself ($intcontext(Customer') = Customer'$), because $I(Customer') \neq \lambda$. The interpreted context of attribute `name'` of class `Customer'` is class `Customer'` as well ($intcontext(name') = Customer'$), because `Customer'` is the closest ancestor class to `name'` which has an interpretation. And, for the same reason, the interpreted context of association connecting classes `Customer'` and `Partner'` is class `Customer'`. On the other hand, class `Contact'` does not have an interpretation ($I(Contact') = \lambda$). The closest ancestor class with an interpretation is class `Customer'`. Therefore, $intcontext(Contact') = Customer'$. Similarly, $intcontext(ItemTester') = intcontext(ItemPricing') = Item'$. And the same is for attributes, for example $intcontext(tester') = Item'$.

Note that $intcontext(X')$ may be empty, i.e. $intcontext(X') = \lambda$. In that case we will say that X' does not have an interpreted context. Thus, having the notion of interpreted context, we are ready to introduce the rules.

We now define the notion of *consistent interpretation* of a PSM schema against a PIM schema. Consistency ensures that the semantics of the PSM schema determined by the interpretation is consistent with the semantics modeled by the PIM schema.

Definition 4.7. Let I be an interpretation of a PSM schema S' against a PIM schema S . We say that I is *consistent* if the following rules are satisfied:

$$(\forall C' \in S'_c \text{ s.t. } repr(C') \neq \lambda \wedge I(C') \neq \lambda)(I(C') = I(repr(C'))) \quad (1)$$

$$(\forall A' \in S'_a \text{ s.t. } I(A') \neq \lambda)(intcontext(A') \neq \lambda \wedge I(A') \in attributes(I(intcontext(A')))) \quad (2)$$

$$(\forall R' \in S'_r \text{ s.t. } I(child(R')) = \lambda \vee I(intcontext(R')) = \lambda)(I(R') = \lambda) \quad (3)$$

$$(\forall R' \in S'_r \text{ s.t. } I(child(R')) \neq \lambda \wedge intcontext(R') \neq \lambda) \quad (4)$$

$$(I(R') = (E_1, E_2) \text{ s.t. } participant(E_1) = I(intcontext(R')) \wedge participant(E_2) = I(child(R')))$$

Condition (1) requires that a structural representative C' of a class $repr(C')$ has the same interpretation as $repr(C')$. This is because C' acquires the attributes and content of $repr(C')$. To ensure consistency, the attributes and associations in the content must semantically remain with C .

Condition (2) requires that when an interpreted attribute A' has an interpreted context C' , then $I(A')$ must be an attribute of $I(C')$. In other words, A' must semantically belong to the interpretation of its interpreted context.

Conditions (3) and (4) ensure consistency of associations. Condition (3) requires that only an association with an interpreted child and interpreted context may have an interpretation. This is because the semantics of an association specifies how instances of the child of the association are connected to their interpreted context. For associations with interpretation, condition (4) is applied. It is similar to (2). If an association R' has an interpreted context with interpretation C and its child has an interpretation D , the interpretation of R' must be an ordered image of an association connecting C and D .

Let us demonstrate conditions (2)–(4) on the PSM schema depicted in Fig. 5(a). First, suppose attribute `tester'`. Its interpreted context is class `Item'` with $I(Item') = Item$. Condition (2) requires that $I(tester') \in attributes(Item)$. This is satisfied in our case because $I(tester') = tester$. Second, suppose the association connecting classes `Customer'` and `Contact'`. Since $I(Contact') = \lambda$, condition (3) requires that the association does not have an interpretation. This is natural, because both `Contact'` represents a part of class `Customer` from the conceptual perspective and, therefore, it is meaningless to specify the semantics of the association. On the other hand, the association connecting classes `Customer'` and `Partner'` must have an interpretation, because both classes have an interpretation and it is necessary to specify the semantics of the connection between them. The interpretation must be an association connecting `Customer` and `Partner` according to condition (3). In our case it is the association *responsibility* which is correct.

The following lemma shows that Definition 4.7 is correct.

Lemma 4.1. Let I be a consistent interpretation of a PSM schema S' against a PIM schema S . The semantics of each component of S' specified by I is unambiguous.

Proof. We will show that the semantics of each PSM class, attribute or association in S' is specified unambiguously by I . Without loss of generality, we will consider components of S' which are semantically related to a PIM class $C \in S_c$.

First, let $C' \in S'_c$ s.t. $I(C') = C$. The semantics of C' is specified by I unambiguously from Definition 4.5. There is no way to use I to deduce that the semantics of C' is a class $C_0 \neq C$.

Second, let $A' \in S'_a$ s.t. $I(A') \neq \lambda$. Let $C'_a \in S'_c$ s.t. $C'_a = class(A')$ or $repr(C'_a) = class(A')$. Let $I(intcontext(C'_a)) = C$. If $I(A') \notin attributes(C)$, the semantics of A' is ambiguous. From the conceptual perspective, A' semantically belongs to C on one hand and it does not on the other. However, $I(A') \notin attributes(C)$ cannot happen because of conditions (1) and (2).

Third, let $R' \in S'_r$ s.t. $I(R') \neq \lambda$ is a directed image of an association $R \in S_r$. Let $C'_r \in S'_c$ s.t. $C'_r = parent(R')$ or $repr(C'_r) = parent(R')$

or $C'_r = \text{child}(R')$ (in this last case, condition (4) ensures that $I(C'_r) \neq \lambda$ and, therefore, $\text{intcontext}(C'_r) = C'_r$). Let $I(\text{intcontext}(C'_r)) = C$. If $R \notin \text{associations}(C)$, the semantics of R' is ambiguous. From the conceptual perspective, R' is an association connected to C on one hand and it is not on the other. However, $R \notin \text{associations}(C)$ cannot happen because of conditions (1) and (3). \square

In the rest of this paper, each interpretation considered will be consistent; we do not consider inconsistent interpretations. In the following section, we introduce atomic operations which allow for modifications of PIM and PSM schemas. It is clear that the consistency of an interpretation may be corrupted when the PIM or PSM schema is changed. For example, removing an attribute in the PIM schema may break condition (2) of Definition 4.7 and reconnecting an association in the PSM schema may break conditions (3) and (4). Our aim is to introduce a mechanism which propagates performed changes from the PIM schema to the PSM schema and vice versa so that the consistency of the interpretation is ensured.

5. Atomic operations

In this section, we introduce *atomic operations* for editing PIM and PSM schemas. They are not intended to be used directly by the designer, because they are too primitive and using them would be too laborious and clumsy for the designer. However, they will serve us as a formal basis for describing more user-friendly operations composed of these atomic operations. In Section 7 we will describe *composite operations*. In Section 6 we will describe how operations are propagated between PIM and PSM levels to ensure the consistency of corrupted interpretations.

Formally, we suppose a PIM schema $\mathcal{S} = (S_c, S_a, S_r)$ and a set of PSM schemas $\mathcal{PSM} = \{S'_1, \dots, S'_n\}$, where each S'_i has an interpretation I_i against \mathcal{S} . We also consider one specific PSM schema $S' = (S'_c, S'_a, S'_r, S'_m, C'_s)$ from this set with an interpretation I against \mathcal{S} . For each atomic operation, we specify its input parameters together with a precondition and postcondition. If a precondition is not satisfied, the operation cannot be performed. The postcondition describes the effect of the operation. When an operation is executed on S or S' , we say that the schema *evolved to a new version*. This is denoted S^+ or S'^+ , respectively. The new version of the interpretation will be denoted I^+ . Initially, we suppose a single empty PIM schema and empty \mathcal{PSM} . The PIM schema cannot be removed. On

the other hand, a new PSM schema with an interpretation against the PIM schema may be created and later removed.

We classify atomic operations into 4 categories: *creation* (denoted by the Greek letter α), *update* (denoted by the Greek letter ν), *removal* (denoted by the Greek letter δ) and *synchronization* (denoted by the Greek letter σ). While the creation, update and removal operations are common in the literature, the synchronization operations have not been considered and are novel in our approach. They are crucial for the evolution.

A synchronization operation allows for the specification that two sets of components are *semantically equivalent*. Consider a simple scenario with a class *Customer* which models a concept of customer. Customer's address is modeled with attribute *address*. Later, users require a more precise specification of address including street, street number and city. Therefore, the designer needs to replace *address* with new attributes *street*, *streetno* and *city*. According to existing approaches, this means creating the new attributes and removing the old one. However, this leads to losing the information that the old attribute is semantically equivalent to the new set. Without this information, the performed change cannot be correctly propagated as we will show later. This is the reason why we propose synchronization operations. We use them to specify that *address* is semantically equivalent to the set $\{\textit{street}, \textit{streetno}, \textit{city}\}$.

Definition 5.1. Let \mathcal{X}_1 and \mathcal{X}_2 be two sets of components from the same PIM or PSM schema. We use predicate $\textit{equiv}(\mathcal{X}_1, \mathcal{X}_2)$ to denote that \mathcal{X}_1 and \mathcal{X}_2 are *semantically equivalent*. It means that \mathcal{X}_1 models the same information as \mathcal{X}_2 .

5.1. Atomic operations for PIM schema evolution

We start with atomic operations for evolution of PIM schemas. The operations for creating new components are summarized in Table 2: their semantics is clear and so we provide no further description. Let us just note that the name, data type, and cardinality of created components are set to default values configured by the schema designer.

The operations for updating components are summarized in Table 3. There are two update operations which merit a more detailed explanation – moving an attribute A from its current class to another class ($\nu_a^{\textit{class}}$) and reconnecting an association end E from its current class to another class ($\nu_r^{\textit{class}}$). The preconditions of both

Table 2
Atomic operations for creating new PIM components.

Notation	Description	Precondition	Postcondition
$C = \alpha_c()$	Create class C with default name l_c	true	$C \in (S_c^+ \setminus S_c) \wedge \textit{name}^+(C) = l_c$
$A = \alpha_a(C)$	Create attribute A with default name, type and cardinality l_a, l_t , and l_c	$C \in S_c$	$A \in (S_a^+ \setminus S_a) \wedge \textit{class}^+(A) = C \wedge \textit{name}^+(A) = l_a$ $\wedge \textit{type}^+(A) = t_a \wedge \textit{card}^+(A) = c_a$
$R = \alpha_r(C_1, C_2)$	Create association R with default name and cardinalities l_r and c_r	$C_1, C_2 \in S_c$	$R = \{E_1, E_2\} \in (S_r^+ \setminus S_r) \wedge \textit{name}^+(R) = l_r$ $\wedge \textit{participant}^+(E_1) = C_1 \wedge \textit{participant}^+(E_2) = C_2$ $\wedge \textit{card}^+(E_1) = c_r \wedge \textit{card}^+(E_2) = c_r$

Table 3
Atomic operations for updating PIM components.

Notation	Description	Precondition	Postcondition
$\nu_c^{\textit{name}}(C, v)$	Update name of class to v	$C \in S_c$	$\textit{name}^+(C) = v$
$\nu_a^{\textit{name type card}}(A, v)$	Update name, type, or cardinality of attribute to v	$A \in S_a$	$\textit{name}^+(A) = v, \textit{type}^+(A) = v$ or $\textit{card}^+(A) = v$
$\nu_a^{\textit{class}}(A, C)$	Move attribute to class C	$A \in S_a \wedge C \in S_c \wedge \textit{associations}(\textit{class}(A), C) \neq \emptyset$	$\textit{class}^+(A) = C$
$\nu_r^{\textit{name}}(R, v)$	Update name of association to v	$R \in S_r$	$\textit{name}^+(R) = v$
$\nu_r^{\textit{class}}(E, C)$	Reconnect association end to class C	$C \in S_c \wedge (\exists R \in S_r)(E \in R) \wedge \textit{associations}(\textit{participant}(E), C) \neq \emptyset$	$\textit{participant}^+(E) = C$
$\nu_r^{\textit{card}}(E, v)$	Update cardinality of association end to v	$(\exists R \in S_r)(E \in R)$	$\textit{card}^+(E) = v$

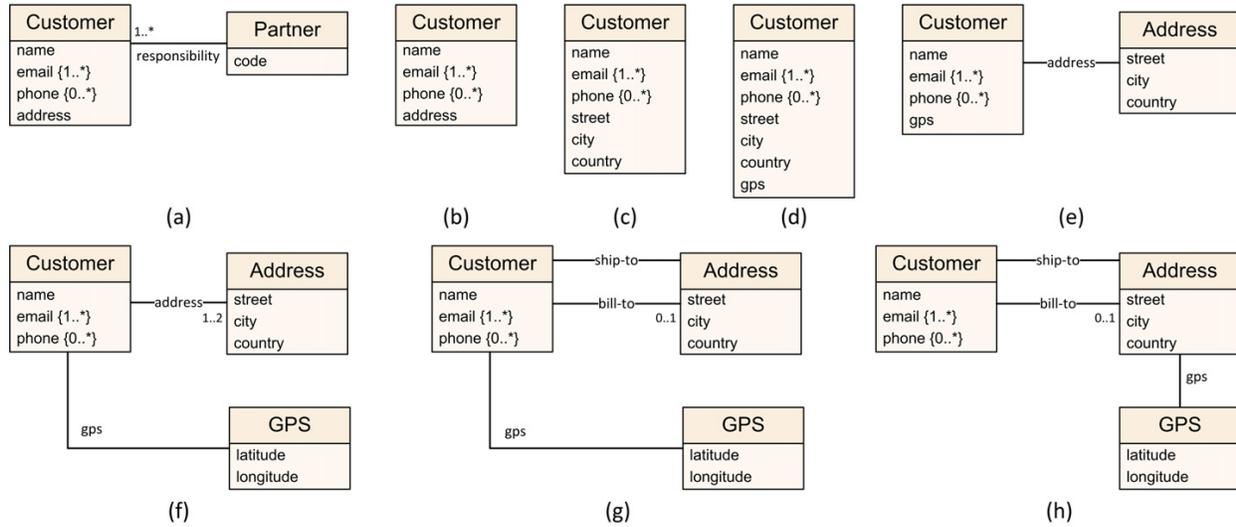


Fig. 7. Evolution of a sample PIM schema demonstrating the introduced creation, update, removal and synchronization atomic operations.

Table 4
Atomic operations for removing PIM components.

Notation	Description	Precondition	Postcondition
$\delta_c(C)$	Remove class C	$C \in \mathcal{S}_c \wedge \text{attributes}(C) = \emptyset \wedge \text{associations}(C) = \emptyset$	$C \notin \mathcal{S}^+$
$\delta_a(A)$	Remove attribute A	$A \in \mathcal{S}_a$	$A \notin \mathcal{S}^+$
$\delta_r(R)$	Remove association R	$R \in \mathcal{S}_r$	$R \notin \mathcal{S}_c^+$

operations require that the current and new class are connected by an association ($\text{associations}(C_1, C_2)$ denotes all associations connecting classes C_1 and C_2). Therefore, it is not possible to move an attribute or reconnect an association end between classes which are only connected by a path of associations or are not connected at all. However, it is possible to create a composite operation from the atomic operations which allows for moving the attributes and reconnecting association ends freely. It can perform atomic moves or reconnections in case where there is a connecting path. And, it can create a temporary association connecting the classes in case there is no connection at all.

The operations for removing components are summarized in Table 4; however, the class removal operation (δ_c) requires the removed class to have no attributes and connected associations, so all attributes and associations connected to the class must be removed before removing the class itself.

And, finally, the operations for synchronizing components are summarized in Table 5. We introduce two operations – synchronization of two sets of attributes and synchronization of two sets of associations. The precondition of the former synchronization operation requires the attributes from both sets to belong to the same class. It is not restrictive. It is possible to have two synchronized sets of attributes, where each attribute is in a different class. However, we need to perform a sequence of atomic operations – this consists of moving the attributes to the same class, synchronization and

moving them back to their original classes. Similarly, the precondition of the other operation needs the associations to connect the same two classes. Again, it is not restrictive, because other cases may be achieved by performing a sequence of atomic operations.

The reader might notice that we do not provide an operation for synchronizing classes. An operation for synchronizing a mixture of classes, attributes and associations is missing as well. Our preliminary case studies (one of the provided in Section 9) show that class synchronization is not necessary as classes do not model data but only encapsulate them. Synchronization of a mixture of components would be, theoretically, necessary, but too complex and unnatural for common designers. Therefore, in the current version of our technique we try to manage the evolution without these advanced synchronization operations. We leave this scientifically interesting issue to our future work.

A sample evolution is depicted in Fig. 7. Fig. 7(a) shows a starting PIM schema. It is a fragment of the PIM schema depicted in Fig. 4. It contains two classes *Customer* and *Partner* which model customers and partners, respectively. Partners are responsible for customers which is modeled by the relationship *responsibility*. First, there is a requirement to not further consider partners. Therefore, class *Partner* needs to be deleted by operation $\delta_c(\text{Partner})$. It is necessary to perform $\delta_a(\text{code})$ and $\delta_r(\text{responsibility})$, which delete the attribute *code* and association *responsibility*, prior to $\delta_c(\text{Partner})$. The result is depicted in Fig. 7(b).

Table 5
Atomic operations for synchronization of PIM components.

Notation	Description	Precondition	Postcondition
$\sigma_a(\mathcal{X}_1, \mathcal{X}_2)$	Synchronize set of attributes \mathcal{X}_2 with set of attributes \mathcal{X}_1	$\mathcal{X}_1 \subseteq \mathcal{S}_a \wedge \mathcal{X}_2 \subseteq \mathcal{S}_a$ $\wedge (\exists C \in \mathcal{S}_c)(\mathcal{X}_1, \mathcal{X}_2 \subseteq \text{attributes}(C))$	$\text{equiv}^+(\mathcal{X}_1, \mathcal{X}_2)$
$\sigma_r(\mathcal{X}_1, \mathcal{X}_2)$	Synchronize set of associations \mathcal{X}_2 with set of associations \mathcal{X}_1	$\mathcal{X}_1 \subseteq \mathcal{S}_r \wedge \mathcal{X}_2 \subseteq \mathcal{S}_r$ $\wedge (\exists C_1, C_2 \in \mathcal{S}_c)(\mathcal{X}_1, \mathcal{X}_2 \subseteq \text{associations}(C_1, C_2))$	$\text{equiv}^+(\mathcal{X}_1, \mathcal{X}_2)$

Table 6
Atomic operations for creating PSM schemas and their components.

Notation	Description	Precondition	Postcondition
$(S', I) = \alpha'_s(S)$	Create new PSM schema S' with interpretation I against S	true	$S' = (\{C'_s\}, \emptyset, \emptyset, \emptyset, C'_s) \wedge$ $S' \in (PSM^+ \setminus PSM) \wedge I = \{(C'_s, \lambda)\}$
$C' = \alpha'_c()$	Create new class C' with default name l_c	true	$C' \in (S'_c^+ \setminus S'_c) \wedge I^+(C') = \lambda$ $\wedge name^+(C') = l_c$
$A' = \alpha'_a(C')$	Create new attribute A' with default name, type, XML form and cardinality l_a, t_a, x_a , and c_a	$C' \in S'_c$	$A' \in (S'_a^+ \setminus S'_a) \wedge class^+(A') = C'$ $\wedge I^+(A') = \lambda \wedge name^+(A') = l_a$ $\wedge type^+(A') = t_a \wedge xform^+(A') = x'_a$ $\wedge card^+(A') = c_a$
$R' = \alpha'_r(X'_1, X'_2)$	Create new association R' with default name and cardinalities l_r and c_r	$X'_1 \in (S'_c \cup S'_m)$ $\wedge X'_2 \in (S'_c \cup S'_m) \setminus \{C'_s\}$ $\wedge (\exists R_0)(child(R_0) = X'_2)$	$R' \in (S'_r^+ \setminus S'_r) \wedge parent(R') = X'_1$ $\wedge child(R') = X'_2 \wedge I^+(R') = \lambda$ $\wedge name^+(R') = l_r \wedge card^+(R') = c_r$ $\wedge position(R')^+ = content(C') $
$M' = \alpha'_m()$	Creates new sequence content model M'	true	$M' \in (S'_m^+ \setminus S'_m) \wedge$ $cmttype(M') = sequence$

Second, there is a requirement to consider customer's addresses in more detail. Currently, it is modeled by attribute *address* of class *Customer*. The aim is to model addresses as depicted in Fig. 7(h). The evolution is iterative. Particular iterations are depicted in Fig. 7(c)–(g). The designer starts with modeling addresses with three separate attributes *street*, *city* and *country* instead of the original attribute *address*. For this, (s)he creates the attributes ($street = \alpha_a(\text{Customer}, \dots)$), changes the default values of their names and data types when necessary ($v_a^{name}(street, "street", \dots)$), synchronizes them with the original attribute ($\sigma_a(\{address\}, \{street, city, country\})$) and, finally, removes the original attribute ($\delta_a(address)$). The synchronization is important. It specifies that the new attributes are semantically equivalent with the old one. The whole sequence of performed atomic operations can be viewed as splitting the original attribute into the three new ones. Note that the precondition for synchronization is satisfied (all attributes are in the same class). The result is depicted in Fig. 7(c).

Later, the designer notices that (s)he forgot to include GPS information. (S)he needs to extend the three attributes *street*, *city* and *country* with a new attribute *gps*. For this, (s)he creates the new attribute and synchronizes the original set of attributes modeling address with the new set which is the original set extended with *gps* ($\sigma_a(\{street, city, country\}, \{street, city, country, gps\})$). The result is depicted in Fig. 7(d).

Now, class *Customer* contains too much information and the designer wants to make it more transparent. Therefore, (s)he decides to move attributes *street*, *city* and *country* to a separate class *Address*. The class is not present and (s)he, therefore, needs to create it and update its name ($Address = \alpha_c()$, $v_c^{name}(Address, "Address")$). (S)he also needs to connect it with *Customer* by creating a new association *address* ($address = \alpha_r(\text{Customer}, Address, \dots)$). Then, (s)he can move the attributes to the new class ($v_a^{class}(street, Address, \dots)$). The old and new class are connected by an association and, therefore, the precondition for moving the attributes is satisfied. The result is depicted in Fig. 7(e). (S)he also needs to detail *gps* to latitude and longitude and move them to a separate class *GPS*. Therefore, she performs a similar sequence of operations as for the former *address* attribute. And, finally, (s)he needs to extend customers to have one or two addresses instead of one. (S)he, therefore, changes the cardinality of association *address* to 1..2 ($v_r^{card}(address2, 1..2)$), where *address2* is the endpoint associated with *Address*. The result is depicted in Fig. 7(f).

In the following step, the designer gets a requirement to explicitly distinguish the semantics of the two addresses to a mandatory shipping address and optional billing address. Therefore, (s)he splits the association *address* to two new associations *shipto* and *billto*. As with the splitting of attributes, this

Table 7
Atomic operations for updating PSM components.

Notation	Description	Precondition	Postcondition
$v_c^{name}(C', v)$	Update name of class C' to v	$C' \in S'_c$	$name^+(C') = v$
$v_c^{repr}(C', C'_r)$	Set class C' as structural representative of C'_r	$C' \in S'_c \setminus \{C'_s\} \wedge (C'_r = \lambda \vee (C'_r \in S'_c \setminus \{C'_s\} \wedge I(C') = I(C'_r) \wedge C' \neq repr^+(C'_r)))$	$repr^+(C') = C'_r$
$v_m^{cmttype}(M', t)$	Update type of content model M'	$M' \in S'_m \wedge t \in \{sequence, choice, set\}$	$cmttype^+(M') = t$
$v_a^{name type}(A', v)$	Updates name, type, cardinality, or XML form of attribute to v	$A' \in S'_a$	$name^+(A') = v$, $type^+(A') = v$, $card^+(A') = v$ or $xform^+(A') = v$
$v_a^{pos}(A')$	Changes position of attribute A' by -1	$position(A') > 1$ $A' \in S'_a \wedge C' \in S'_c \wedge$ $(repr(class(A')) = C' \vee class(A') = repr(C') \vee$ $class(A') = parentclass(C') \vee$ $C' = parentclass(class(A')))$	$class^+(A') = C'$
$v_r^{name card}(R', v)$	Update name or cardinality of association R' to v	$R' \in S'_r$	$name^+(R') = v$ or $card^+(R') = v$
$v_r^{pos}(R')$	Change position of association R' by -1	$position(R') > 1$	$position^+(R') =$ $position(R') - 1$
$v_r^{class}(R', P')$	Reconnect parent association end of association R' to new parent P'	$R' \in S'_r \wedge P' \in S'_c \cup S'_m \wedge$ $(repr(parent(R')) = P' \vee parent(R') = repr(P') \vee$ $\exists R_0 \in S'_r \text{ which connects } parent(R') \text{ and } P')$	$parent^+(R') = P'$

Table 8
Atomic operations for updating interpretations.

Notation	Description	Precondition	Post...
$\mathcal{V}_c^{int}(C', C)$	Update interpretation of class C' to class C	$C' \in \mathcal{S}_c' \setminus \{C'_s\} \wedge (C = \lambda \vee C \in \mathcal{S}_c) \wedge$ $(\forall A' \in \mathcal{S}_a \text{ s.t. } \text{intcontext}(A') = \text{intcontext}(C') \wedge C' \in \text{anc}(A'))(I(A') = \lambda) \wedge$ $(\forall R' \in \mathcal{S}_r' \text{ s.t. } (\text{intcontext}(R') = \text{intcontext}(C') \wedge C' \in \text{anc}(R')) \vee \text{child}(R') = C')$ $(I(R') = \lambda) \wedge$ $(\forall C'_0 \in \mathcal{S}_c' (\text{repr}(C'_0) \neq C') \wedge \text{repr}(C') = \emptyset)$	$I'(C') = C$
$\mathcal{V}_a^{int}(A', A)$	Update interpretation of attribute A' to attribute A	$A' \in \mathcal{S}_a' \wedge (A = \lambda \vee (A \in \mathcal{S}_a \wedge \text{class}(A) = I(\text{intcontext}(A'))))$	$I'(A') = A$
$\mathcal{V}_r^{int}(R', O)$	Update interpretation of association R' to directed image O of association R	$R' \in \mathcal{S}_r' \wedge \text{child}'(R') \in \mathcal{S}_c' \wedge$ $O = \lambda \vee (O = (E_1, E_2) \wedge$ $\text{participant}(E_1) = I(\text{intcontext}(R')) \wedge \text{participant}(E_2) = I(\text{child}(R'))$ $)$	$I'(R') = O$

entails creating two new associations ($\text{shipto} = \alpha_r(\text{Customer}, \text{Address}, \dots)$), changing their default names and cardinalities ($\mathcal{V}_r^{\text{name}}(\text{shipto}, \text{"shipto"}, \dots)$), synchronizing the old association with the new ones ($\sigma_r(\{\text{address}\}, \{\text{shipto}, \text{billto}\})$), and removing the old association ($\delta_r(\text{address})$). Note that the synchronized associations connect the same classes and, therefore, preconditions for the synchronization are satisfied. The result is depicted in Fig. 7(g).

Finally, there appears a requirement to record GPS information for each address instead of a customer. For this, the designer reconnects the association gps from class Customer to class Address ($\mathcal{V}_r^{\text{class}}(\text{gps1}, \text{Address})$), where gps1 is the endpoint associated with Customer . Again, the precondition for the reconnection is satisfied, because the classes are connected by an association. The result is depicted in Fig. 7(h).

5.2. Atomic operations for PSM schema evolution

In this section, we introduce atomic operations for evolution of PSM schemas. The operations for creating new components are summarized in Table 6: there is also an operation for creating PSM schemas themselves. Again, names, data types, XML forms and cardinalities of new components are set to default values which are configured by the designer. All components are created with an empty interpretation against the PIM schema.

The operations for updating components are summarized in Table 7. Similar to the operations for updating PIM components, there are two interesting operations – moving an attribute ($\mathcal{V}_a^{\text{class}}$) and reconnecting an association end ($\mathcal{V}_r^{\text{class}}$). Both are similar to their PIM equivalents but there are some differences. An attribute can be moved to the nearest ancestor or descendant class of its current class ($\text{parentclass}(C')$ denotes the nearest ancestor class to C'). It can also be moved to a structural representative of its current class or, conversely, to a class which is a structural representative of its current class. For an association, only its parent association end can be reconnected to the parent or to any child of its current parent. When its current parent is a class, it can also be reconnected

to a structural representative of the current parent or, conversely, to a class which is a structural representative of its current parent.

The operations for updating interpretations are summarized in Table 8. Their preconditions ensure that the consistency of interpretation is not violated. Concretely, the operation for updating class interpretation (\mathcal{V}_c^{int}) could violate any of the conditions necessary for consistency. However, its precondition requires that the interpretation of any attribute or association, whose consistency would be corrupted by the update, must be empty. This includes all attributes and associations which have the same interpreted context as the class ($\text{anc}(X')$ used in the precondition which denotes all ancestor classes of X'). Also, the class can not be a structural representative and, conversely, it cannot have a structural representative. Therefore, the conditions can not be violated.

The operation for updating attribute interpretation (\mathcal{V}_a^{int}) could affect condition (2) and the operation for updating association interpretation (\mathcal{V}_r^{int}) could affect conditions (3) and (4). Their preconditions prevent any violations. (They are directly rewritten from the definition.)

The operations for removing components of PSM schemas are listed in Table 9. Their functionality is quite clear. Let us note that we can only remove classes and content models that are empty and are roots of their PSM schema. Also, we can only remove associations, whose removal does not violate Definition 4.7. When there are attributes or associations in the subtree of R' with the same interpreted class context as R' and with non-empty interpretations, we cannot remove R' . To correct the schema, we would need to set empty interpretations to these attributes and associations, which is not an atomic operation. Note that when we remove an association going to a class or content model, this class or content model becomes a root.

And, finally, the operations for synchronizing two sets of PSM components are listed in Table 10. Similar to their PIM equivalents, they allow for synchronization of two sets of attributes and two sets of associations. The operation for attributes corresponds to its PIM equivalent. The operation for associations is also similar. However, it is not possible to require the associations to have the same

Table 9
Atomic operations for removing PSM schemas and their components.

Notation	Description	Precondition	Post...
$\delta_c(S')$	Remove existing PSM schema S' and its interpretation I against \mathcal{S}	$S' \in \text{PSM} \wedge S' = (S'_c, S'_a, S'_r, S'_m, C'_s)$ $\wedge S'_a = S'_r = S'_m = \emptyset \wedge S'_c = \{C'_s\}$	$S' \notin \text{PSM}^+$
$\delta_c(C')$	Remove class C'	$C' \in \mathcal{S}_c' \wedge \text{attributes}(C') = \text{content}(C') = \emptyset \wedge (\nexists C'_0 \in \mathcal{S}_c' (\text{repr}(C'_0) = C'))$	$C' \notin \mathcal{S}^+$
$\delta_a(A')$	Remove attribute A'	$A' \in \mathcal{S}_a'$	$A' \notin \mathcal{S}^+$
$\delta_r(R')$	Remove association R'	$R' \in \mathcal{S}_r' \wedge$ $(\forall X' \in (\mathcal{S}_a \cup \mathcal{S}_r : \text{intcontext}(X') = \text{intcontext}(R') \wedge R' \in \text{anc}(X'))(I(X') = \lambda)$	$R' \notin \mathcal{S}^+$
$\delta_m(M')$	Remove cont. model M'	$M' \in \mathcal{S}_m' \wedge \text{content}(M') = \emptyset$	$M' \notin \mathcal{S}^+$

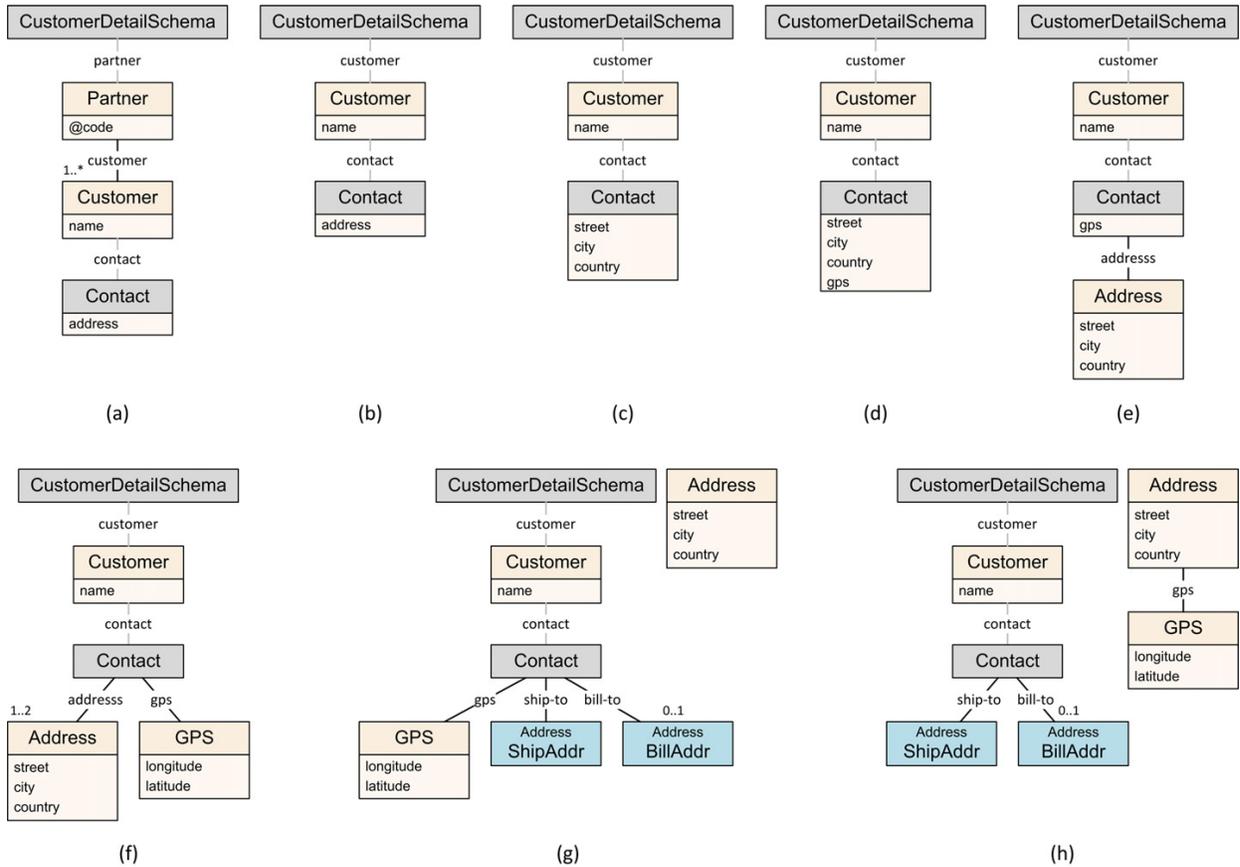


Fig. 8. Evolution of a sample PSM schema demonstrating the introduced creation, update, removal and synchronization atomic operations.

participants (because of the tree nature of PSM schemas). Instead, we require that they have one of their participants in common: that is the child of none or one of the associations and the parent of the others. The other participants must be different classes but with the same non-empty interpretation. In other words, these other participants are semantically equivalent (they have the same class in the PIM schema as their interpretation). Therefore, the operation also corresponds to its PIM equivalent.

Note that similar to synchronization in a PIM schema, the expression $equiv^+(\mathcal{X}'_1, \mathcal{X}'_2) = true$ in the postconditions of both operations denotes that \mathcal{X}'_1 and \mathcal{X}'_2 are synchronized in the new version of the PSM schema.

We demonstrate the operations in Fig. 8. Fig. 8(a) shows a starting PSM schema. It has an interpretation against the PIM schema depicted in Fig. 7(a). The PIM schema evolves as we have

demonstrated, so the consistency of the interpretation of the PSM schema is broken. In this example, we show how the PSM schema and its interpretation can be adapted using the introduced atomic operations to ensure the consistency. Fig. 8(b)–(h) show particular evolutionary steps which result from changes at the PIM level demonstrated in Fig. 7.

First, the designer needs to remove class `Partner'` ($\delta'_c(Partner')$) to reflect the first change in the PIM schema (removing class `Partner`). Prior to this, (s)he removes attribute `code'` ($\delta'_a(code')$) and both associations `partner'` and `customer'` ($\delta'_r(partner')$, $\delta'_r(customer')$). Then, (s)he creates a new association `customer'` connecting the schema class and class `Customer'` ($customer' = \alpha'_r(CustomerDetailSchema', Customer')$) and sets its name ($\nu_r^{name}(customer', "customer")$). The association has an empty interpretation. The result is depicted in Fig. 8(b).

Table 10
Atomic operations for synchronization of components of PSM schemas.

Notation	Description	Precondition	Postcondition
$\sigma'_a(\mathcal{X}'_1, \mathcal{X}'_2)$	Synchronize set of attributes \mathcal{X}'_2 with set of attributes \mathcal{X}'_1	$\mathcal{X}'_1 \subseteq S'_a \wedge \mathcal{X}'_2 \subseteq S'_a$ $\wedge (\exists C' \in S'_c)(\mathcal{X}'_1, \mathcal{X}'_2 \subseteq attributes(C'))$	$equiv^+(\mathcal{X}'_1, \mathcal{X}'_2)$
$\sigma'_r(\mathcal{X}'_1, \mathcal{X}'_2)$	Synchronize set of associations \mathcal{X}'_2 with set of associations \mathcal{X}'_1	$\mathcal{X}'_1 \subseteq S'_r \wedge \mathcal{X}'_2 \subseteq S'_r \wedge$ $(\exists C'_1 \in S'_c, C'_2 \in S_c \cup \{\lambda\})(\forall R' \in \mathcal{X}'_1 \cup \mathcal{X}'_2)($ $(C'_1 = parent(R') \wedge child(R') \in S'_c \wedge$ $(I(child(R')) = C_2)) \vee$ $(C'_1 = child(R') \wedge parent(R') \in S'_c \wedge$ $(I(parent(R')) = C_2)))$	$equiv^+(\mathcal{X}'_1, \mathcal{X}'_2)$

Second, `address` was split into three new attributes in the PIM schema. The designer correspondingly needs to split attribute `address'` into three new attributes `street'`, `city'`, and `country'` in the PSM schema. (S)he creates the attributes ($street' = \alpha'_a(Contact')$, ...) and sets their names ($\nu_a^{name}(street', "street")$, ...) and interpretations ($\nu_a^{int}(street', street)$, ...). Then, (s)he synchronizes the new attributes with the original one ($\sigma'_a(\{address'\}, \{street', city', country'\})$). Note that the preconditions of both setting interpretations and synchronization is satisfied – the attributes are in the respective interpreted context and are within the same class. Finally, (s)he removes the old attribute `address'` ($\delta'_a(address')$). The result is depicted in Fig. 8(c).

The designer then proceeds with extending the three new attributes with a new attribute `gps'`. (S)he creates the attribute ($gps' = \alpha'_a(Contact')$, ...) and sets its name ($\nu_a^{name}(gps', "gps")$, ...) and interpretation ($\nu_a^{int}(gps', gps)$, ...). Then, (s)he specifies that the three original attributes are semantically equivalent to the extension ($\sigma'_a(\{street', city', country'\}, \{street', city', country', gps'\})$). The result is depicted in Fig. 8(d).

Third, the designer needs to separate attributes `street'`, `city'`, and `country'` to a new class `Address'`. (S)he creates the new class ($Address' = \alpha'_c()$) and sets its name ($\nu_c^{name}(Address', "Address")$) and interpretation ($\nu_c^{int}(Address', Address)$). Then, (s)he connects the new class with `Contact'` by creating a new association ($address' = \alpha'_r(Contact', Address')$). (S)he sets its name ($\nu_r^{name}(address', "address")$) and interpretation ($\nu_r^{int}(address', address)$). Now, the preconditions allow for moving the attributes from `Contact'` to `Address'` ($\nu_a^{class}(street', Address')$, ...). The designer moreover needs to specify that a customer has one or two addresses ($\nu_r^{card}(address', 1..2)$). The result is depicted in Fig. 8(e). Later, (s)he similarly moves the attribute `gps'` to a new class `GPS'` and splits it into two new attributes `longitude'` and `latitude'` as depicted in Fig. 8(f).

Fourth, the PIM schema now distinguishes two different addresses – shipping and billing. The designer needs to reflect this change in the PSM schema by splitting the association `address'` correspondingly. However, the change is more complex than in case of the PIM schema, because the resulting PSM schema must be a tree. (S)he first needs to create new classes `ShipAddr'` and `BillAddr'`, set their names, and set their interpretation to PIM class `Address`. Now (s)he may split `address'`. (S)he creates two new associations `shipto'` and `billto'` connecting `Contact'` with `ShipAddr'` and `BillAddr'`, respectively. (S)he sets their names and interpretations to PIM associations `shipto` and `billto`, respectively. (S)he also sets cardinality of `billto'` to 0..1. Then, (s)he synchronizes the original association with the new ones ($\sigma'_r(\{address'\}, \{shipto', billto'\})$) and removes the original one ($\delta'_r(address')$). (S)he wants both new addresses to model the same XML fragments as the original one and (s)he, therefore, sets `ShipAddr'` and `BillAddr'` as structural representatives of `Address'` ($\nu_c^{repr}(ShipAddr', Address')$, ...).

In the final step, the designer needs to reflect in the PSM schema reconnecting the association `gps` in the PIM schema. The impact of this change to the PSM schema is that both, shipping and billing address have GPS information. Therefore, the designer needs to reconnect `gps'` association to class `Address'`. This requires two atomic reconnections of `gps'`. First, from class `Contact'` to class `ShipAddr'` ($\nu_r^{class}(gps', ShipAddr')$) and then to `Address'` ($\nu_r^{class}(gps', Address')$). Note that both reconnections are allowed by the operation precondition. In the first case the reconnection is between classes connected by an association. In the other case, the reconnection is between a structural representative and its referenced class.

6. Propagation of atomic operations

According to Section 4.3, an interpretation of a PSM schema S' against a PIM schema S must be consistent. When S or S' is modified by an atomic operation, one or more conditions necessary for consistency may be violated and, consequently, the interpretation or the other schema must be adapted accordingly. We call the process which ensures the adaptation *propagation of the atomic operation*. In the example in the previous section we showed how a designer can solve this issue manually (our designer performed a sequence of operations in the PIM schema and then (s)he needed to perform similar steps in the PSM schema). In this section, we show how the propagation can be automated. If we consider the fact that there may be many PSM schemas affected, automation is very helpful.

6.1. Propagation from PIM to PSM level

In this section, we describe how introduced atomic operations executed on the PIM schema S are propagated to each PSM schema $S' \in \mathcal{PSM}$ and its interpretation I against S . We will demonstrate the propagation on our sample PIM and PSM schema evolution depicted in Figs. 7 and 8, respectively. We suppose that the designer manually changes the PIM schema in the steps depicted in Fig. 7. In Section 5.2 we showed in Fig. 8 how the designer manually adapts the PSM schema according to the changes in the PIM schema. In this section we show that our propagation mechanism is able to adapt the PSM schema automatically which reduces the designer's manual work.

Let us start with propagating the creation operations. Creating a new component X in S does not automatically imply the existence of any component in S' . This is because the creation does not violate Definition 4.5. Moreover, X models a new part of the reality which has no representation in the PSM schemas, where its creation could be propagated. It is up to the designer, whether to create new components in the PSM schemas which represent this new part of the reality, or not. Therefore, the creation operations are not propagated.

Let us consider the evolution of our sample PIM schema depicted in Fig. 7(e). Here, the designer first created a new class `Address` and association `address`. These operations on their own do not automatically result in creating new classes and associations in the PSM schemas. It is up to the designer whether to propagate them, or not. For example, (s)he later decides to move some attributes from `Customer` to `Address`. In that case it is necessary to create new classes with `Address` as their interpretation in the PSM schemas, because we need to correspondingly move attributes in the PSM schemas.

An update of a component X of S may have an impact on each component X' in the PSM schema with $I(X') = X$ and its propagation may be necessary. More specifically, an update of the name of X is propagated to an optional update of the name of X' . This is because X and X' do not necessarily need to share the same name. On the other hand, an update of the type or cardinality of X is propagated to a mandatory update of the type or cardinality of X' .

In our sample PIM schema evolution depicted in Fig. 7(f), the cardinality of the association endpoint of association `address` connected to class `Address` was updated from 1..1 to 1..2. This is automatically propagated by our mechanism to all associations in the PSM schemas with `address` as an interpretation. For example, it is propagated to association `address'` depicted in Fig. 8(f).

The propagation of the two remaining update operations, i.e. moving an attribute and reconnecting an association end, is more complex. Both operations modify the structure of S which may break the consistency of the interpretation. The impact on the structure of S' may be quite extensive and it would be almost impossible for the designer to manage the impact manually. The

idea of propagation is similar for both operations even though reconnecting an association end is technically more complicated. However, we will discuss only the first one.

Suppose that $v_a^{class}(A, D)$ was performed. In other words, an attribute A in the PIM schema was moved from its current class C to another class D . Consider an attribute A' in the PSM schema s.t. $I(A')=A$. Since the interpretation is consistent, we see that $class(A')=C'$ s.t. $intcontext(C')=C=class(A)$. By executing $v_a^{class}(A, D)$ we get $class(A)=D$. We see that, on one hand, A' is semantically an attribute of C . On the other hand, we see that the semantics is A which is an attribute of $D \neq C$.

Therefore, the move of A must be propagated to a corresponding move of A' . Concretely, we have to move A' to a class with an interpretation D to make the interpretation consistent. The move and its propagation is illustrated in Fig. 9. Fig. 9(a) contains a PIM schema fragment before executing the operation (on the left hand side of the thick arrow) and the fragment after the move (on the right hand side). Fig. 9(b) and (c) contain three PSM schema fragments before and after the propagation. They illustrate three basic situations which may occur.

Suppose class C' in the PSM schema with interpretation C . Let A' be an attribute of C' with an interpretation A . The first situation is depicted in Fig. 9(b). Here, C' contains an association R' with an interpretation R . Its child is class D' with an interpretation D . In this case the propagation means moving A' to D' which makes the interpretation consistent. The second situation is depicted in Fig. 9(c). Here, there is an association R' with an interpretation R which goes to C' . Its parent is class D' with an interpretation D . Again, this case means moving A' to D' . The last situation is depicted in Fig. 9(d). Here, there is no association connected to C' and with an interpretation R . In this case, propagation means creating a new association R' with an interpretation R connecting C' and a new class D' with an interpretation D . A' may be again moved to D' . Also there are some other situations which differ from the three demonstrated only in technical details. This includes situations with content models or classes without an interpretation on the path between C' and D' . We have solved these situations in our implementation but do not specify them in this paper.

In a general case, there can be more and different associations R_1, \dots, R_n connecting C and D . There are associations R'_1, \dots, R'_n connected to C' with directed images of R_1, \dots, R_n as interpretations, respectively. If some R'_i is missing, we ask a designer if it should be created.³ If we apply the previous idea, we get up to $C'_{v,1}, \dots, C'_{v,n}$ classes, where A' should be moved. However, such move is not possible. Instead, we make a copy A'_i of A' for each $C'_{v,i}$ and move the copy to $C'_{v,i}$. Making a copy means the following sequence of atomic operations: (1) creating A'_i , (2) synchronizing it with A' (it is important since it specifies that A' and A'_i model the same information), (3) setting the properties of A'_i to the same values as A' , and (4) moving A'_i .

In our sample evolution depicted in Fig. 7(e), the designer moved attributes `street`, `city` and `country` from class `Customer` to class `Address`. This makes the interpretation of the PSM schema depicted in Fig. 8(d) inconsistent. There are attributes `street'`, `city'` and `country'` having the moved PIM attributes as their interpretation. Our propagation mechanism ensures automatically that the attributes are moved correspondingly so that the interpretation is consistent again as depicted in Fig. 8(e). First, the mechanism automatically creates a new class `Address'` which was not present in the PSM schema and connects it with class `Contact'` by a new association. Then, it automatically moves the attributes.

Removing components of S must be propagated by removing corresponding components of S' or setting their interpretations to λ to keep the interpretation consistent. More specifically, removing an attribute A leads to removing each attribute A' in S' s.t. $I(A')=A$ or setting $I(A')=\lambda$. Both solutions are correct (i.e. they do not break the consistency of interpretation) and, therefore, the designer has to decide. Removing an association leads mandatorily to removing each association R' in S' s.t. $I(R')$ is a directed image of R . We cannot set $I(R')=\lambda$. This is because condition (3) of Definition 4.7, R' with a non-empty interpretation has a child with a non-empty interpretation and vice versa. Setting $I(R')$ to λ would break this condition. And, finally, removing a class C leads to removing each class C' in S' s.t. $I(C')=C$ or setting $I(C')$ to λ . Both possibilities are correct. From the precondition of the operation for removing a class, C has no attributes and there are no associations connected to C . Because of conditions (2) and (3) of Definition 4.7, there is no attribute or association in S' in the interpreted context of C' with a non-empty interpretation. Therefore, it is possible to set $I(C')=\lambda$. It is also possible to remove C' . However, it may have attributes and there may be associations connected to C' with empty interpretations. These must be removed first. There are also some technical details we do not discuss further. For example, parent ends of the associations going from C' may be reconnected to the parent of C' in certain cases etc.

In our sample evolution depicted in Fig. 7(b), the designer removed association `responsibility` and class `Partner` with its attribute `code`. The propagation mechanism ensured that the corresponding components in our sample PSM schema were removed after a dialogue with the designer as depicted in Fig. 8(b).

Synchronizing two sets \mathcal{X}_1 and \mathcal{X}_2 of components of S means that the existence of both sets must be synchronized at all levels. Whenever there is an equivalent to \mathcal{X}_1 in the PSM schema S' there must also be an equivalent to \mathcal{X}_2 and vice versa.

The operation for synchronization of attributes, i.e. $\sigma_a(\mathcal{X}_1, \mathcal{X}_2)$, only enables one to synchronize two sets of attributes which have a common class C . Let C' be a class s.t. $I(C')=C$ which contains attributes whose interpretations are all attributes from \mathcal{X}_1 . Since \mathcal{X}_1 and \mathcal{X}_2 are semantically equivalent, our propagation mechanism interprets this as a fact that C' must be supplemented with new attributes so that it contains attributes whose interpretations are all attributes from \mathcal{X}_2 as well (and conversely) (there are some technical details we do not discuss in more detail). First, we have to consider also all attributes of $repr(C')$ if $repr(C') \neq \lambda$. Second, we have to consider all attributes with C' as an interpreted context, not only the attributes of C' .

In our sample evolution depicted in Fig. 7(c), the designer split the original attribute `address` into three new attributes `street`, `city` and `country`. For this, after the creation of the new attributes (which is not propagated to the PSM level as we have already discussed), the designer synchronized the original attribute with the new ones. The synchronization is automatically propagated by our mechanism as follows: wherever there is an attribute `address'` with interpretation `address` in a PSM schema, create three new attributes `street'`, `city'` and `country'` and synchronize them with `address'`. The result of this automatic propagation is depicted in Fig. 8(c). Note that after the synchronization, the designer removed the original attribute `address`. This was propagated by our mechanism to removing the attribute `address'` in the PSM schema after the decision of the designer.

The operation for synchronization of associations, i.e. $\sigma_r(\mathcal{X}_1, \mathcal{X}_2)$, is very similar to the previous case. Again, when their common class C' contains associations whose interpretations are directed images of all associations from \mathcal{X}_1 , it must be supplemented so that it contains associations whose interpretations are directed images of all associations from \mathcal{X}_2 , and vice versa. Again, there are technical details we omit for space limitations, i.e. we must not forget those

³ If all of them are missing and the designer decides not to create any, no propagation is performed.

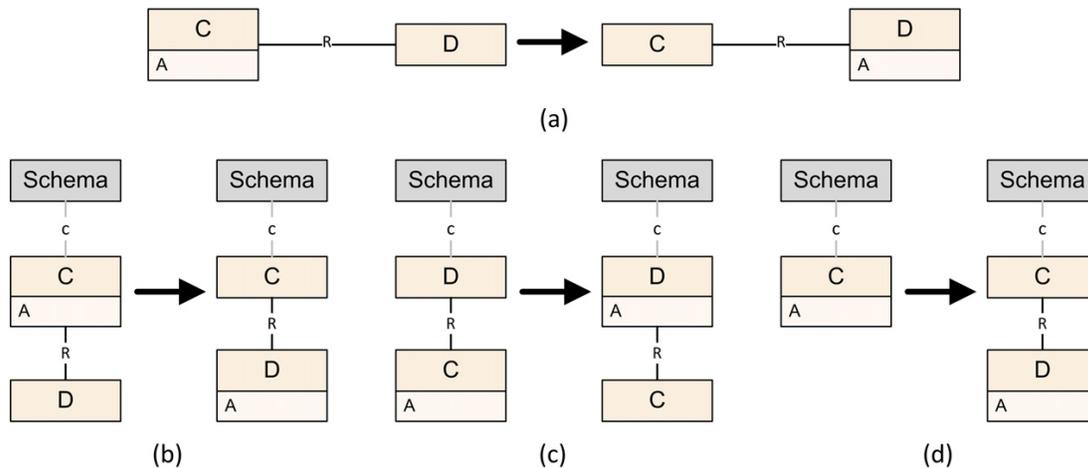


Fig. 9. Visualization of the mechanism for propagating the operation for moving PIM attributes.

associations that are implicitly in the content of C' – it is a structural representative and we must consider not only associations which have C' as a parent but all which have C' as their interpreted context.

Synchronization of associations is demonstrated in Fig. 7(g), where the designer split association *address* into two new associations *shipto* and *billto*. The result of automatic propagation is depicted in Fig. 8(g).

6.2. Propagation from PSM to PIM level

In this section, we describe the opposite direction of propagation, i.e. how operations executed on a PSM schema $S' \in \mathcal{PSM}$ are propagated to the PIM schema S . Again, we will demonstrate the propagation on our sample PIM and PSM schema evolution depicted in Figs. 7 and 8, respectively. Now, we will, however, suppose that the designer manually changes the PSM schema according to the steps depicted in Fig. 8. We show how our propagation mechanism ensures that the PIM schema is adapted automatically.

Creating a new component in S' does not directly imply an existence of any component in S and, therefore, creation operations are not propagated from PSM to the PIM level. For example, when creating a new class *Address'* and association *address'*, connecting *Contact'* and *Address'* in Fig. 8(e) does not imply creating a corresponding class and association in the PIM schema. The creation will be performed in the PIM schema only when it is explicitly required by the designer. The propagation mechanism then also ensures that interpretations of the class and association in the PSM schema are set correctly. The result is depicted in Fig. 7(e).

Updating components in S' has an effect on corresponding components in S with some exceptions; there are updates with no effect on the PIM schema S , because the updated properties have no equivalent in S . This includes updating a structural representative of a class, updating the position or XML form of an attribute and updating the position of an association. There are also updates which are only optionally propagated to S . This is similar to the other direction; for example, changing a name of an attribute. And, there are operations which are propagated mandatorily. The simple case is, for example, updating a cardinality which is propagated in a straightforward manner. And, as in the other direction, there are two operations whose propagation is mandatory and more complex: moving an attribute and reconnecting an association end.

When the interpretation of a moved attribute or reconnected association is empty, the change is not propagated at all to S . This is because the updated component has no equivalent in S

and, therefore, consistency of interpretation is not broken. Similarly, no propagation is necessary when the interpreted context of the updated component has not changed. In that case, there is no change from the conceptual point of view.

For example, suppose the PSM schema in Fig. 8(b). Let the designer move attribute *address'* from class *Contact'* to class *Customer'*. The move is within the same interpreted context (which is class *Customer'*) and, therefore, the attribute was not moved from the conceptual perspective and its interpretation remains consistent. No propagation is necessary in this case.

In other cases, propagation is necessary. However, except for the technical details, the principles of the propagation are similar to the other direction and so, we omit their detailed description. For example, suppose that the designer moves attributes *street'*, *city'* and *country'* from class *Contact'* to class *Address'* as depicted in Fig. 8(e). The interpreted context is changed (from class *Customer'* to class *Address'*). Our propagation mechanism automatically ensures that the interpretations of the three attributes (i.e. *street*, *city* and *country*) are moved correspondingly in the PIM schema. The resulting PIM schema is depicted in Fig. 7(e).

Similar to the updates, removing a component from S' is not propagated to S when the removed component has an empty interpretation. Removing a PSM component X' with an interpretation X may imply removing X when there are no other PSM components with interpretation X . However, even when there are no PSM components with interpretation X , we do not remove X' automatically. This is because PSM schemas are only views of the whole domain modeled by the PIM schema. Absence of a given concept modeled by X in the views does not imply the necessity of removing X from the PIM schema. The removal of X is, therefore, only optional.

For example, when the designer removes class *Partner'* as depicted in Fig. 8(b), the propagation mechanism asks the designer whether the corresponding class *Partner* in the PIM schema should be removed as well, or not. In our case, the designer decides to remove *Partner* as depicted in Fig. 7(b).

Synchronization of two sets \mathcal{X}'_1 and \mathcal{X}'_2 is propagated from S to S' very similarly as in the opposite direction. The only difference is that there are components in \mathcal{X}'_1 and \mathcal{X}'_2 with and without an interpretation. If \mathcal{X}'_1 (or \mathcal{X}'_2) contains only components with an interpretation, its semantic equivalent exists in S and each component X' of \mathcal{X}'_2 (or \mathcal{X}'_1), respectively, which does not have an interpretation is, therefore, propagated to S . Propagation means creating a new component X corresponding to X' and setting $I(X')=X$. Otherwise, the synchronization is not propagated to the

PIM level, because an equivalent of \mathcal{X}'_1 or (or \mathcal{X}'_2 , respectively) does not exist in \mathcal{S} .

Sample synchronization operations are demonstrated in Fig. 8(c) (attribute synchronization) and Fig. 8(g) (association synchronization). They are automatically propagated by our mechanism to the PIM schema as depicted in Fig. 7(c) and (g), respectively.

6.3. Minimality and correctness of atomic operations

Important properties of any set of atomic operations are their minimality and correctness. Minimality means that there is no atomic operation which could be expressed as a sequence of other atomic operations. Correctness means that the proposed operations are correct. In our specific case it means not only that an atomic operation transforms a schema from a consistent state to another consistent state but also that the propagation mechanism preserves the consistency of interpretations of PSM schemas to PIM schemas.

Theorem 6.1. *The set of atomic operations is minimal.*

Proof. Assume the operations for evolution of classes in the PIM schema, i.e. α_c , δ_c and ν_c^{name} . Without α_c we are not able to create any class. Similarly, without δ_c we are not able to remove any class. Finally, without ν_c^{name} we are not able to change the class name. It cannot be set during the creation, because α_c sets a default name. The proof for other atomic operations for creating, removing and updating PIM and PSM components is similar. The operations for synchronizing two sets of attributes or associations are clearly atomic as well. No other operation allows for synchronization. \square

Theorem 6.2. *The set of atomic operations together with the propagation mechanism is correct.*

Proof. We have already proved the correctness in the previous text. In Section 5.2 we have shown that the preconditions of operations for updating interpretations of PSM components ensure that the consistency of interpretation can not be broken. In Sections 6.1 and 6.2 we have shown that the propagation mechanism repairs the consistency of interpretation when broken by moving attributes, reconnecting associations ends and removing components. We have also shown that the other operations do not touch the consistency at all. And, finally, we have shown in these sections that whenever the propagation mechanism needs to perform a sequence of atomic operations to repair the consistency of interpretation, the preconditions of these operations are always satisfied so that the sequence may be performed in any time. \square

6.4. Completeness of atomic operations

Sometimes completeness is understood as a property which ensures that for any two given schemas there always exists a sequence of atomic operations which transform one of the schemas to the other. The sequence usually removes all components of the former schema and creates the components of the other. This is not a correct proof of completeness, because it does not consider possible semantic relationships between the components of both schemas. The old components are simply removed and the new ones are created without preserving the semantic relationships. However, this only covers the structural part of the schema. What we also aim for is preserving the semantic part of the schemas. This is largely dependent on the user and his/her interpretation of the meaning of the schemas. However, even if semantic relationships are considered (e.g. semantic equivalence in our case) it is not easy to prove general completeness formally. Even though such proof would be interesting from the theoretical point of view, it is beyond the scope of this paper. Instead, our aim is to demonstrate completeness practically. In this paper we provide a case study of

real world system, where we applied our approach. It experimentally shows that the proposed set of atomic operations is complete. The case study can be found in Section 9.

7. Composite operations

The atomic operations introduced formally in the previous sections were proposed so that they form minimal and correct set as proven above. Naturally, they are not supposed to be used directly by the user in all cases and it is not the whole set of available operations. In this section we show examples how the atomic operations can form more user-friendly and realistic *composite* operations.

Formally, a composite operation is a sequence of two or more atomic operations. As we have shown in the previous text, propagation mechanism ensures that any atomic operation does not corrupt the consistency of affected interpretations. Therefore, composition of atomic operations preserves consistency as well and it is not necessary to extend the propagation mechanism with specifics of the composition.

Creation with parameters. A simple composite operation necessary in every system is creating a particular component with pre-set values. We show such case in the operation $createPIMAttr(C, n, t, c)$ which allows for creating of a PIM attribute in a class C with name n , data type t and cardinality c . It consists of the following steps:

$$A = \alpha_a(C); \nu_a^{name}(A, n); \nu_a^{type}(A, t); \nu_a^{card}(A, c)$$

The propagation mechanism optionally creates corresponding PSM components.

Splitting of a PIM attribute. This operation is a typical example of *drill-down modeling*, i.e. creating more and more precise data structures. An example of such an operation is shown in Fig. 7(c), where the designer needs to detail a single-valued address of a customer to street, city and country. In general, the composite operation $splitPIMAttr(A, \{n_1, n_2, \dots, n_k\})$ for splitting a PIM attribute A of a class C to a set of attributes with names $\{n_1, n_2, \dots, n_k\}$ consists of the following steps:

$$\begin{aligned} A_1 &= createPIMAttr(C, n_1, type(A), card(A)); \dots; A_k \\ &= createPIMAttr(C, n_k, type(A), card(A)); \\ \sigma_a(\{A\}, \{A_1, A_2, \dots, A_k\}); \delta_a(A) \end{aligned}$$

The propagation mechanism, in particular in case of synchronization, ensures that all the PSM attributes representing A are replaced with PSM attributes representing A_1, A_2, \dots, A_k . In our sample depicted in Fig. 7(c), the designer would execute a single composite operation $splitPIMAttr(address, \{ "street", "city", "country" \})$.

Removing a PSM tree. In the previous case we have shown an example of a composite operation which consists of a sequence of atomic operations and a composite operation which consists of atomic and other composite operations. Operation $removePSMtree(C')$ for removing a PSM tree rooted at class C' is an example of a recursive composite operation, i.e. it calls itself if necessary. The operation consists of the following steps:

1. $(\forall R' \in content(C')) \text{ removePSMtree}(child(R'))$;
2. $(\forall A' \in attributes(C')) \delta_a(A')$;
3. $(\forall R'_p \in S'_r \text{ s.t. } child(R'_p) = C') \delta_r(R')$;
4. $\delta_c(C')$;

Naturally, we cannot provide the full list of possible composite operations, as the particular set depends on the choice of the vendor of a particular system and the requirements of users. Our aim was

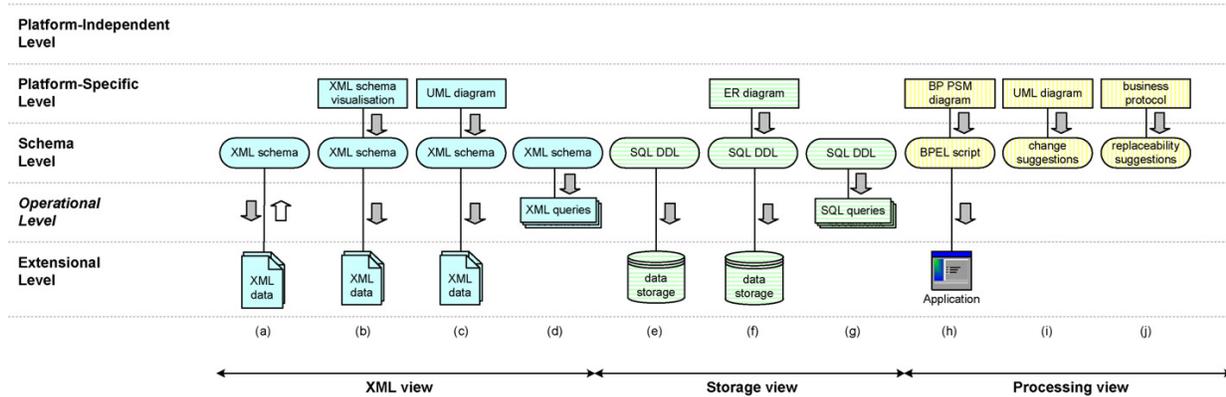


Fig. 10. Current XML evolution approaches.

to demonstrate that the proposed mechanism can be used in real-world situations.

8. Related work

The current approaches towards evolution management can be classified according to distinct aspects (Mens and Van Gorp, 2006; Czarnecki and Helsen, 2006). The changes and transformations can be expressed (OMG, 2005; Boronat et al., 2006) as well as divided (Cicchetti et al., 2009) variously too. However, to our knowledge there exists no general framework comparable to our proposal in Section 3; particular cases and views of the problem have previously only been solved separately, superficially and mostly imprecisely without any theoretical or formal basis. In this section we describe the closest and most advanced approaches related to our proposal.

XML view. We can divide the current approaches to XML schema evolution and change management into several groups as depicted in Fig. 10. Approaches in the first group (a) consider changes at the schema level and differ in the selected XML schema language, i.e. DTD (Al-Jadir and El-Moukaddem, 2003; Coox, 2003) or XML Schema (Tan and Goh, 2005; Cavalieri, 2010). In general, the transformations can be variously classified. For instance, paper (Tan and Goh, 2005) proposes *migratory* (e.g. movements of elements/attributes), *structural* (e.g. adding/removal of elements/attributes) and *sedentary* (e.g. modifications of simple data types). The changes are expressed variously and more or less formally. For instance in Cavalieri (2010) a language called *XUpdate* is described. The changes are then automatically propagated to the extensioal level to ensure validity of XML data. There also exists an opposite approach that enables one to evolve XML documents and propagate the changes to their XML schema (Bouchou et al., 2004). Approaches in the second (b) and third (c) group are similar, but they consider changes at an abstraction of logical level – either visualization (Klettke, 2007) or a kind of UML diagram (Domínguez et al., 2005). Both cases work at the PSM level, since they directly model XML schemas with their abstraction. No PIM schema is considered. All approaches consider only a single separate XML schema being evolved.

Another open problem related to schema evolution is adaptation of the respective XML queries, i.e. propagation to the operational level (Fig. 10(d)). Unfortunately, the amount of existing works is relatively low. Paper (Moro et al., 2007) gives recommendations on how to write queries that do not need to be adapted for an evolving schema. On the other hand, in Geneves et al. (2009) the authors consider a subset of XPath 1.0 constructs and study the impact of XML schema changes on them.

In all the papers cited the authors consider only a single XML schema. In Passi et al. (2009) multiple *local* XML schemas are considered and mapped to a *global* object-oriented schema. Then, the authors discuss possible operations with a local schema and their propagation to the global schema. However, the global schema does not represent a common problem domain, but a common integrated schema; the changes are propagated just upwards and the operations are not defined rigorously. The need for well defined set of simple operations and their combination is clearly identified in Section 6 of a recent survey of schema matching and mapping (Bellahsene et al., 2011).

Storage view. The idea of evolution and change management in XML storage strategies is currently focused particularly on data updates and, usually, joined with *XQuery Update Facility* (Chamberlin et al., 2007). However, this is not the area we are dealing with since the updates are mostly considered within the respective XML schema. As depicted in Fig. 10(e), in the area of evolution of general database schemas we can find approaches that focus on evolution of (object-)relational schemas (Curino et al., 2009, 2008) as well as object-oriented schemas (Banerjee et al., 1987; Lerner, 2000). Similar to the case of XML schema evolution, there are also approaches that deal with propagation from an ER schema, i.e. PSM level, to a relational schema (An et al., 2008b), i.e. schema level (Fig. 10(f)) or propagation to an operational level (Curino et al., 2009) (Fig. 10(g)).

In the purely XML-related approaches we need to consider *schema-driven* storage strategies. As surveyed in Simanovsky (2008), the amount of the respective approaches is not high. We can find first attempts of change propagation in the current leading object-relational database management systems – *Oracle DB*,⁴ *IBM DB2*⁵ and *Microsoft SQL Server*.⁶ In this case we can differentiate two types of schema evolution – whether *backward compatibility* of the changes, i.e. preservation of data validity, is required, or not. Both the DB2 and SQL Server require the backward compatibility. Oracle DB also supports change propagation regardless backward compatibility; however, it is not done automatically; a data expert must provide an XSLT script which re-validates the stored XML documents. To ease this approach we have recently proposed an algorithm that enables one to provide such transformation script semi-automatically (Malý et al., 2011).

Processing view. Since we are considering the area of evolution of XML applications, we cannot omit the most popular applica-

⁴ <http://www.oracle.com/us/products/database/>.

⁵ <http://www.01.ibm.com/software/data/db2/>.

⁶ <http://www.microsoft.com/sqlserver/2008/en/us/>.

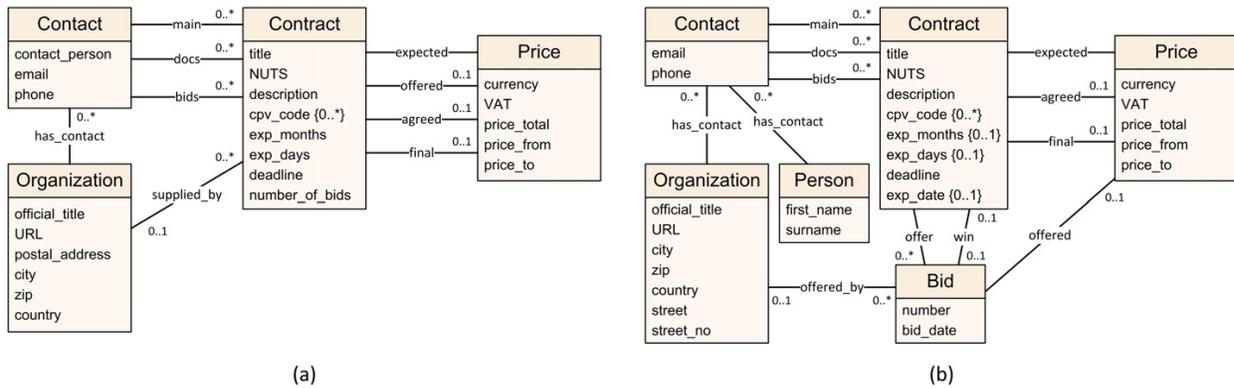


Fig. 11. (a) PIM schema modeling the NRPP domain, (b) PIM schema evolved according to new requirements.

tion of XML format – Web Services. Currently we can find several approaches that deal with evolution of Web Service; however, again they solve just part of the issues described (ai Sun et al., 2010). In Sindhgatta and Sengupta (2009) the authors describe a plugin to IBM *Rational Software Architect* (RSA)⁷ which enables semi-automatic propagation of changes from business process model of Web Services (Fig. 10(h)) to respective BPEL scripts and thus respective applications. It is one of the frameworks that are very close to our proposal described in Section 3; however, the authors do not provide any theoretical background on the allowed changes or details on the propagation mechanisms. A different approach (Fig. 10(i)) is used in system *Morpheus* (Ravichandar et al., 2008), also based on IBM RSA. At the platform-specific level it considers three UML artifacts – use cases, sequence diagrams and service specifications – and the change propagation among them. The output of the propagation is a set of change suggestions for the respective execution part which should be then done manually by an expert. Similarly, in Ryu et al. (2008) (Fig. 10(j)) the authors deal with change propagation of business protocols of Web Services, i.e. a kind of activity diagrams. The output of the system is a set of recommendations detailing when affected parts are replaceable/migrateable and under what circumstances. Again, the migration is expected to be done by a system expert; however, the system advises how to perform it correctly.

In Andrikopoulos et al. (2008) the authors solve the problem using a completely different strategy. They provide an *abstract service definition model* (ASD) which enables us to model all related concepts of a Web Service, i.e. data structures, behavior and policies at a conceptual level using UML class diagrams. Both ASD and the related operations are defined formally and the completeness and correctness of the operations is proven. On the other hand, change propagation to respective PSMs is not considered and the ASD itself is relatively unnatural. And, considering even more formal approaches and model, in Aversano et al. (2005) the authors model the Web Services using Formal Concept Analysis and, in particular, lattices or in Stevens (2010) using lenses and monoids of edits. However, though the approaches are theoretically very interesting, our aim is to provide less complex and more user-friendly formal background and tools.

9. Case study and evaluation

As has already been mentioned, we have implemented the proposed technique in a tool called *eXolutio* (Klímeček et al., 2011). In

general, it is a proof-of-concept desktop application for conceptual XML data modeling. It implements the PIM and PSM modeling languages and operations for evolution of the PIM and PSM schemas described in this paper. It provides a designer with a set of operations which are composed of the atomic operations described in Section 5. It implements the propagation mechanism introduced in Section 6. At the highest level, *eXolutio* is based on a well known Model View Controller (MVC) design pattern.

Currently, for the purpose of this paper, the atomic operations are implemented in the exact same way they are described here. We use the implementation to experimentally demonstrate that the proposed set of atomic operations is *complete*, i.e. that the atomic operations are sufficient for real-world situations. As we have already discussed in Section 6.4, we do not prove completeness formally in this paper. As to *performance* and *scalability*, it is a fact that a single atomic operation on a PIM schema can lead to a large number of operations in each of the affected PSM schemas. This number can be reduced by some optimizations, improving both performance and scalability. So far, our implementation is strictly based on our *formal model* and focuses on the clear demonstration of our ideas. The issues of performance and scalability will be addressed in later stages of development. It is now clear that some complex operations are far more efficient if they are implemented from scratch, rather than by combining the individual atomic operations. This also holds true for some cases of change propagation. Still, it will always be necessary to prove that the optimized version of the operation has the same formal properties as the non-optimized version would have, which is possible again thanks to our formal model. In addition, many operations are in fact interactive. For example, the designer will choose to which PSM schemas a change will be propagated, in which case the actual time spent by performing the operation will always be comparatively negligible.

In the concluding part of this section we show how the developed technique for designing a family of XML schemas and their evolution on a real-world system was applied. And, finally, we evaluate our technique on the basis of this case study and compare it with other known techniques for XML schema evolution.

9.1. Case study: national register for public procurement

Our case study is the *National Register for Public Procurement* (NRPP)⁸. It is a governmental information system intended for publishing data about public contracts by public authorities in the

⁷ <http://www.01.ibm.com/software/awdtools/architect/swarchitect/>.

⁸ <http://www.isvz.cz> (in Czech only).

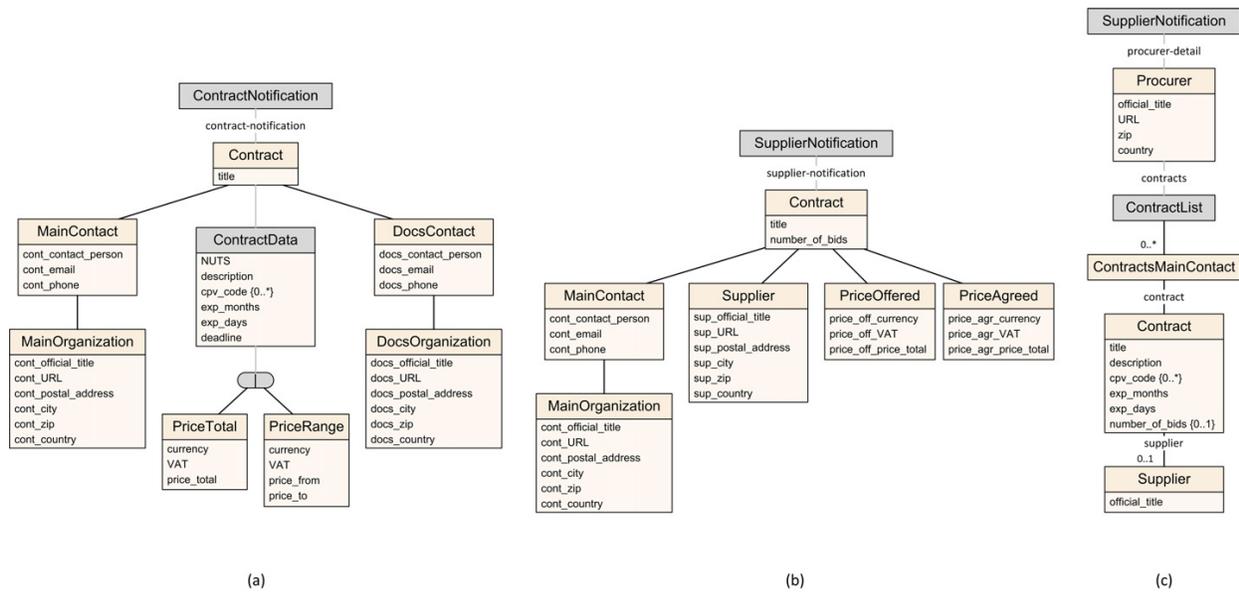


Fig. 12. PSM schemas modeling XML formats for (a) sending contract notifications to NRP, (b) reporting on contract supplier selection to the NRPP, and (c) representing procurer detail.

Czech Republic. Publishing a contract is only obligatory when the contracted price exceeds a level given by the current legislation; otherwise, it is optional. Authorities send contract information formatted according to one of the 17 XML formats accepted by the NRPP. This includes, e.g. XML format for contract notifications, supplier selection notifications, etc.

Currently, the NRPP only provides a textual documentation for the XML formats and a set of sample XML documents. There are neither XML schemas for the XML formats, nor a conceptual schema of the problem domain. Therefore, our first goal was to design not only the XML schemas but also the conceptual schema in a form of a PIM schema and derive PSM schemas for the XML formats from the PIM schema. The resulting PIM schema is depicted in Fig. 11(a). Two of the resulting PSM schemas are depicted in Fig. 12(a) and (b). The PSM schemas are mapped to the PIM schema. The mapping is intuitive and we do not describe it here. The PSM schemas were created exactly according to the textual documentation and XML examples. Let us note that the original schemas we created are more extensive. Due to space limitations, we present here only those parts that bear on our work.

The PIM schema contains classes which model public contracts (class *Contract*) and their procurers and suppliers (class *Organization*). There are also some additional concepts modeled – prices

(class *Price*) and contact information (class *Contact*). There are several relationships modeled with associations. A supplier is associated with a contract by *supplied_by* association. A procurer is associated with a contract by a path of associations *has_contact* and *main*. Each contract has additional contact information – where documentation for the contract is provided (association *docs*) and where bids to the contract are collected (association *bids*). Finally, there are four different prices – expected price (association *expected*), the best offered price (association *offered*), price agreed by a selected supplier and procurer (association *agreed*), and a final real price known after finishing the contract (association *final*).

The PSM schema depicted in Fig. 12(a) models an XML format for notifications about a new public contract. When a public authority issues a new contract, it must send a notification about the contract to the NRPP using this format; it should contain contact information and basic information about the contract. The other PSM schema depicted in Fig. 12(b) models an XML format for notifications about the supplier selected for the contract; it contains the main contract contact, information about the number of offered bids, selected supplier and offered and agreed price.

The numbers of atomic operations executed to create the PIM and PSM schemas are depicted in Fig. 13(a). It shows that only creation and update operations were used.

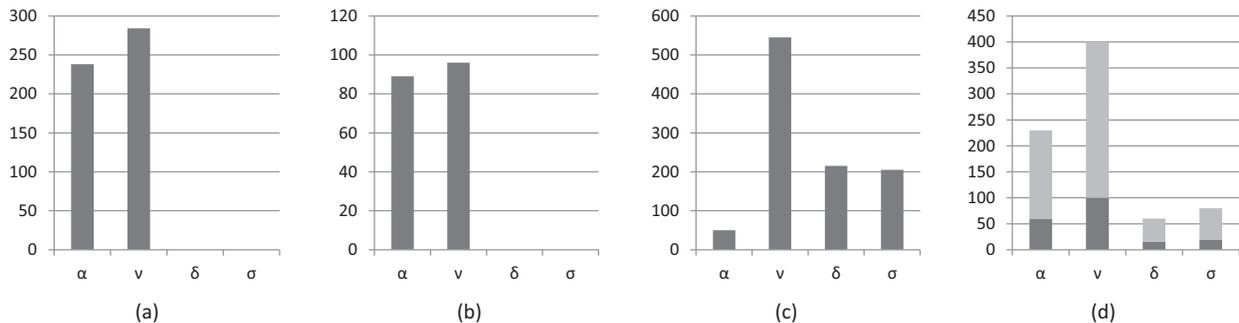


Fig. 13. Numbers of atomic operations performed manually by the designer (dark gray) and automatically by the propagation mechanism (light gray).

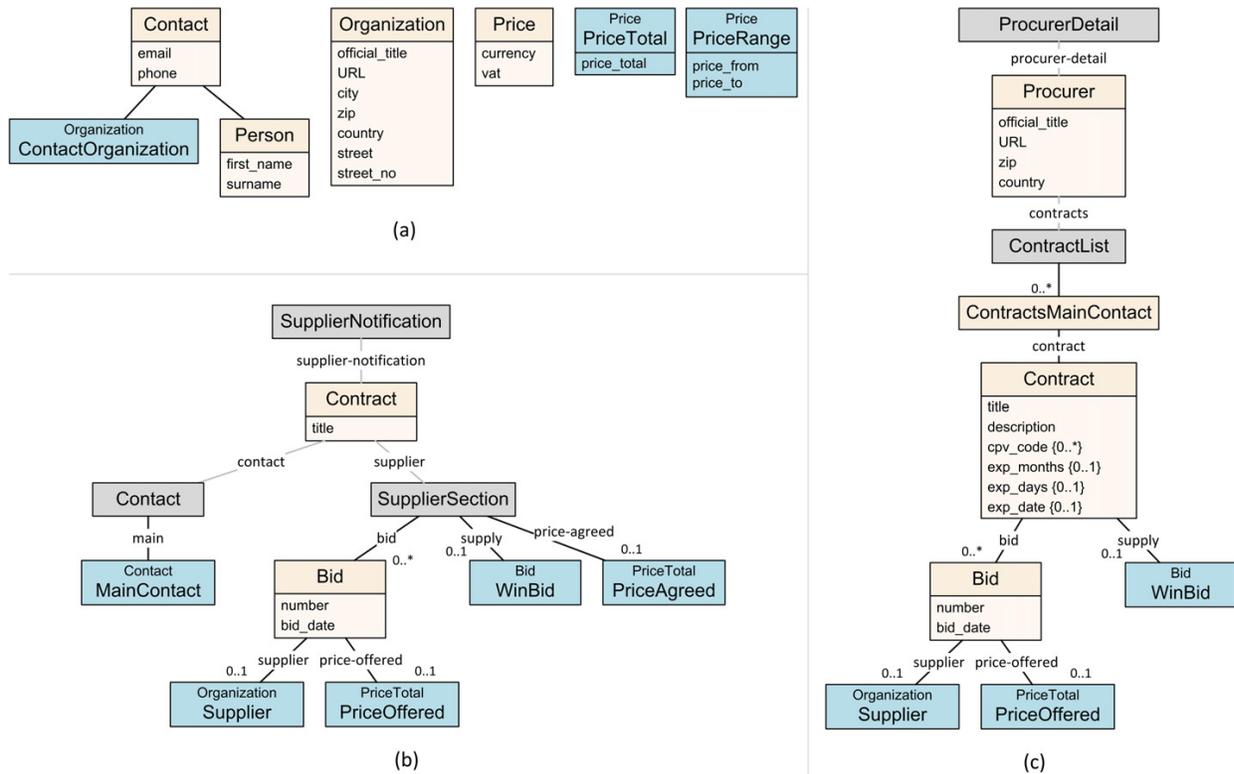


Fig. 14. (a) PSM schema with common components shared between other PSM schemas, (b) evolved PSM schema for reporting on contract supplier selection to the NRPP, (c) evolved PSM schema for representing procurer detail.

There were several issues to solve in this case study. First, the NRPP provides only XML formats which are used by public authorities to send data about their contracts to the NRPP. There are no XML formats for providing information back to the public authorities and other users, e.g. procurer or supplier detail. We show how our approach may be used to easily design such XML formats in a form of PSM schemas on the basis of the existing PIM schema. One such PSM schema which models XML formats for public procurer details is depicted in Fig. 12(c). The numbers of the atomic operations executed at this step are depicted in Fig. 13(b). Again, only the creation and update operations were performed. Even though the designer needs to design the PSM schemas for the new XML formats manually, the experiment clearly showed that our approach saves him/her a great deal of work and prevents him/her from making unnecessary errors. This is because our technique enables us to create the PSM schemas on the basis of the PIM schema (which is quicker than creating PSM schemas separately) and ensures that the designer creates the PSM schemas coherently with the PIM schema (as it preserves the consistency of the interpretation). The designer needs not check whether the PSM schema is semantically correct, or not.

Second, as the reader may have noticed, the quality of the XML formats is low. The designers of the XML formats did not follow basic XML design principles (e.g. exploiting the hierarchical nature of XML); for example, contact information is modeled by XML elements with names prefixed with `cont_`, `docs_`, etc. It would have been better to remove the prefixes and enclose the semantically related XML elements into separate XML elements (e.g. enclose contact XML elements to XML element `contact` structured to `main`, `doc`, etc. or enclose all information related to the supplier into XML element `supplier`). We have made

these adaptations in the present XML formats. Some PSM schema components also appeared which had the same content and we, therefore, used structural representatives to declare the shared content only once. The numbers of the executed atomic operations are depicted in Fig. 13(c). In this step, synchronization and removal operations were also used, because some of the old parts of the PSM schemas were replaced by new ones. Again, the experiment demonstrated that our approach saves a lot of work as it preserves the consistency of PSM schemas against the PIM schema. If the designer makes a change which affects the PIM schema and, possibly, other PSM schemas, our propagation mechanism will notify him/her. We depict the evolved PSM schema from Fig. 12(b) in Fig. 14(b) (it also includes changes described in the following steps). The other PSM schema was evolved similarly. As the reader may see, contact information is now represented hierarchically. Also, the PSM schema is simplified by using structural representatives referring to shared classes contained in a new separate PSM schema depicted in Fig. 14(a).

Third, we implemented various changes which resulted from new requirements on the NRPP functionality and from new legislation. In both cases, changes to the PIM schema needed to be done. The new functionalities required us to model contact persons as a special class instead of attribute `contact_person`. Therefore, we evolved the attribute to a new class `Person` associated with `Contact` and with two new attributes `first_name` and `surname` using our evolution operations.

The new legislation required to report not only the number of bids received for each contract, but also particular bids including the bidding supplier and offered price. Therefore, we replaced the attribute `number_of_bids` with a new class `Bid` with several new attributes. We changed the semantics of `supplied_by` and `offered`

associations by reconnecting them from *Contract* class to the new *Bid* class. Finally, we distinguished the winning bid from the other bids by splitting the association connecting *Bid* and *Contract* classes into two associations *offer* and *win*. The evolved PIM schema is depicted in Fig. 11(b). Since the PIM schema changed, the PSM schemas needed to be adapted accordingly. This was ensured by our propagation mechanism. Fig. 14(b) and (c) show how PSM schemas depicted in Fig. 12(b) and (c) were automatically adapted by the propagation mechanism, respectively.

Finally, there was a requirement to update the XML format for contract notifications (Fig. 12(a)) so that it is possible to give notification not only on the expected months and days in which the contract should be finished, but also on the exact date. Therefore, we added a new attribute *exp.date* which can be used equivalently instead of two present attributes *exp.months* and *exp.years*. This change was correctly propagated to the PIM schema, because it is a conceptual change (see Fig. 11(b)). From here, it was propagated to the other PSM schemas (see Fig. 14(c)).

The numbers of the atomic operations executed during the last two steps are depicted in Fig. 13(d). The darker part shows the numbers of manually executed operations. The lighter part shows the numbers of operations executed automatically by the propagation mechanism.

9.2. Evaluation and comparison to other approaches

The following conclusions are drawn from the above case study:

- All proposed atomic operations are necessary for real-world scenarios as summarized in Fig. 13. The necessity for the creation and updating of atomic operations is clear. The case study showed that we also need removal operations even though we do not want to directly remove parts of data but represent them in more (or less) detailed structures (e.g. splitting attributes). For this, we also need synchronization operations.
- The case study also demonstrates the completeness of the proposed set of atomic operations. Most real-world scenarios we target in our work will be similar to the presented case study (i.e. extending existing schemas with new parts and replacing their existing parts with more (or less) detailed alternatives). For this kind of scenarios our proposed set of operations is complete. On the other hand, there are some limitations. For example, when synchronizing two sets of attributes, we can not exactly specify a function which would transform values between both sets. However, we are not interested in data transformations in this paper but only PIM and PSM schema evolution.
- The existence of the PIM schema and interpretations of PSM schemas against the PIM schema is beneficial when the designer performs creation and update operations for building new PSM schemas or new parts of existing PSM schemas. Our technique ensures that the designer creates new PSM components consistently with the PIM schema (from the conceptual perspective). This ensures semantical coherence between the modeled family of XML schemas. All XML schemas in the family, even those designed by different developers, are consistent with the PIM schema. The designers need not check this coherence manually which saves them a great deal of work and prevents design errors.
- Sometimes the designer may want to optimize the structure of an XML schema but avoid changes to the semantics of the XML schema. When the designer works with the PSM schema, our mechanism is able to prevent these changes. It can automatically check whether a change to the PSM schema needs to be propagated to the PIM schema, or not. This also saves the designer a lot of work, because (s)he does not need to check this manually.
- Finally, when the designer needs to change the PIM schema, our mechanism automatically propagates the changes to the PSM

schemas and vice versa. Again, this saves work and prevents errors, because the designer does not need to propagate the changes manually.

Fig. 13(a)–(d) shows the number of atomic operations performed by the designer in our case study. In comparison to existing approaches to XML schema evolution, our technique saves the designer a great deal of manual work. This is because we consider the PSM schemas interpreted against a single common PIM schema. As we have shown this saves work and prevents errors when the designer needs to check the semantical consistency of his/her new or evolved part of a PSM schema and when making changes to PIM schema or PSM schemas and their propagation to the other schemas. The amount of work saved in comparison to other approaches is demonstrated by Fig. 13. The darker columns show the amount of atomic operations performed manually by the designer. These operations are assisted by our technique which ensures that the consistency between the created XML schemas is preserved. The designer does not need to check consistency manually which saves a lot of time. This consistency check is not provided by existing approaches, where the designer has to do the check manually. The lighter columns show the amount of atomic operations performed automatically by our propagation mechanism. Again, propagation is not supported by existing approaches and these operations would have to be done manually by the designer.

We can also see a fundamental problem in the current approaches, because they do not consider synchronization operations or their equivalent. Without this operation a correct propagation between PIM and PSM schemas is not possible. As we have shown, this is necessary in various practical situations when a part of a PIM or PSM schema is split into more detailed parts. It is also useful in extending an existing part with new components, as well as in a reversed process when more parts of a schema are merged together.

On the other hand, our approach is more laborious in the initial phases, because the PIM schema and PSM schemas modeling the XML schemas must be created. This is not the case of the other approaches which work directly with an XML schema or its direct translations to a conceptual schema. Therefore, the other approaches are more suitable in situations, where the designer works only with a single XML schema. When a family of XML schemas needs to be managed, our approach is more beneficial.

Finally, let us note that the approach presented deals only with PIM and PSM schemas and propagation of changes between both levels. It does not solve the problem of propagation of changes to the data, i.e. XML documents. As we have shown, this has been solved by other approaches. We have also worked on this problem in our previous work. In Malý et al. (2011) we show how XML documents need to be adapted when a PSM schema which models their XML schema evolves.

10. Conclusions

In this paper we focused on two of the main challenges of Model-Driven Development (France and Rumpe, 2007) – evolution and its formal specification. In particular, we were interested in model driven XML schema evolution and concentrated on the PIM and PSM levels of our previously proposed five-level evolution framework. We defined PIM and PSM schemas for modeling XML schemas formally and extended them with atomic and composite operations for their modification. We then identified minimal set of atomic operations, proved its correctness and specified the respective mechanism for automatic propagation of changes between PIM and PSM levels. The formal basis of the operations enables us to ensure that the framework is designed correctly. Next we

introduced implementation of the framework and depicted the advantages of the system in a real-world use case.

Key contributions. If we compare the proposed system with the current approaches, we can identify several key contributions and innovations it brings:

- **Global view of the evolution problem:** As mentioned in Section 8, the existing approaches towards the evolution and change management of XML schemas consider only a single XML schema. Our proposed technique considers a family of XML schemas applied in a system.
- **Formal basis of the proposal:** Similar to the current work being done, we exploit the idea of a platform-independent conceptual schema (PIM schema) of the problem domain which allows for abstraction from technical details of particular XML schemas. We also consider a platform-specific (PSM) schema for each targeted XML schema. Each PSM schema is mapped to the PIM schema and can be automatically translated to an expression in a selected XML schema language such as XSD or RELAX NG. We defined PIM and PSM schemas and mappings between them formally, which enables us to effectively manage the evolution of XML schemas. When a change to the PIM schema is made, we can precisely identify all the parts of XML schemas affected by this modification and, conversely, when a change to the PSM schema is made, we can identify whether the PIM schema is affected and how, or not.
- **Hierarchy of operations:** Naturally, the idea of change management is based on a set of operations. As we have mentioned, they can be classified variously and current approaches utilize different sets. In our work we defined a set of atomic operations and proved its minimality and correctness. Having this concept, we could restrict ourselves to this set and define the respective change propagation precisely and correctly. Last but not least, we showed that using the set of correctly defined atomic operations and the respective change propagation we can define any composite and, hence, more user-friendly operation. The respective change propagation is then defined implicitly and its correctness is ensured as well. A system of operations similar to this was identified in a recent survey (Bellahsene et al., 2011) (Section 6), but has not yet been researched properly.
- **Experimental implementation of the proposal:** The final contribution of this paper is not only the proposal itself, but also its experimental and open-source implementation *eXolutio*. Even though it currently does not cover all the proposed aspects (it is still under intense development), a user may test the key features for his/hers real-world examples. For instance, recently it has been tested in real-world use-cases by the *Fraunhofer Institut*.⁹

Future work. Since the area of XML evolution is relatively new, the number of current approaches and consequently solved issues is not high; there is a significant amount of open problems and future directions we want to focus on. The key areas involve:

- **Specifics of XML schema languages:** In Nečáský and Mlýnková (2010), we show that the introduced PSM is equivalent to the formalism of regular tree grammars (Murata et al., 2005) which are considered as a basic formalism of XML schema languages. However, practical XML schema languages such as (Thompson et al., 2004), introduce various other concepts (e.g. namespaces, use-defined simple data types, etc.) which we did not consider in this paper. We are continuously extending our implementation with these extensions.
- **Other conceptual modeling constructs:** It is common in practice to use other modeling constructs (e.g. inheritance, n-ary relationships, etc.). In our future work, we will deal with these constructs as well.
- **Advanced integrity constraints of the XML data:** As we have mentioned, currently we do not consider integrity constraints that can be expressed using Schematron (Jelliffe, 2001) or XML Schema assertions (Thompson et al., 2004) and we focus on purely grammar-based languages. Our future step is incorporation of advanced integrity constraints into the whole framework. In particular, this involves extension of all levels with advanced constraints, specification of a respective language for the conceptual model and extension of the propagation mechanism. This also includes extending the concept of semantic equivalence with the real semantics (i.e. specification that an address is not only semantically equivalent with street and city but also how). The current approaches (e.g. Xiong et al., 2009) mainly exploit the idea of writing a set of correction rules (mostly using OCL (OCL, 2009) or its extension) that are applied when a particular constraint is violated. Since the language is too general, we will focus on its reasonable subset (Nečáský and Opočenská, 2009; Opočenská and Kopecný, 2008).
- **Operational and extensional level of the framework:** As we have described in Section 3, in this paper we focused on a subpart of the proposed framework – data representation. So, naturally, our next work will focus on those parts which were omitted, especially extensional and operational level. The extensional level is crucial for the applicability of our solution during system runtime. The operational level is also very important and has been mostly omitted in the current literature.
- **Modeling of storage strategies:** Similar to the previous point, in our future work we plan to focus on other parts of the framework which were omitted. An important aspect that has so far not been much considered is the relation of change propagation and XML storage strategies. Currently there are approaches that deal with evolution of database schemas (Curino et al., 2009; Banerjee et al., 1987), but in our case we have to consider a set of applications that form the system, the fact that the XML views of the data can and will overlap and exploitation of the relations between components of the framework. At the same time, we want to preserve the optimal storage strategy for a given application (Mlýnková, 2009).
- **Modeling of business processes:** As we have mentioned, in this text we considered only the modeling of XML data processed and exchanged within an XML system. However, not only the data structures, but also the respective business processes need to be designed, maintained and updated within the evolution process. Our other future aim is to extend and combine the conceptual models of XML data with the respective business processes, to preserve mutual relations and exploit them during the evolution process (Murzek et al., 2006).
- **Relation to ontologies:** An up-to-date and important aspect of data management is establishing and exploiting their semantics. Undoubtedly, the most popular tool for this purpose are currently ontologies. Since an ontology can be viewed as a particular type of schema which has strong relationship to a given XML schema, a natural open issue is developing and maintenance of such relationship under application evolution (Yu and Popa, 2005; An et al., 2008a). However, since the ontologies bear a special type of information, their treatment requires specific approaches (Noy and Klein, 2004).
- **Advanced operations with an XML system:** In this paper we described two types of operations that can occur in an XML system – atomic and composite. However, these are not the only operations that can occur within the system. If we consider the area of integration, we need to deal with the problem of a new

⁹ <http://www.isst.fraunhofer.de/dasinstitut/>.

incoming application and its integration with the current ones (Nguyen et al., 2011), or even integration of a whole XML system. This wide area involves issues such as reverse-engineering of conceptual models and schema matching (Wojnar et al., 2010; Klímek and Nečáský, 2010; Tekli et al., 2009), similarity evaluation (Wojnar et al., 2010) etc.

- *Full implementation of the proposal and efficiency:* Finally, we intend to gradually extend the implemented framework with the proposed improvements. Our aim is to provide a tool that is not only an experimental prototype, but can be applied in complex real-world use cases. Naturally, since our aim is a fully applicable software, we will need to deal with aspects like benchmarking (Alexe et al., 2008) and optimization (Langlois et al., 2006).

Acknowledgements

This work was partially supported by the Czech Science Foundation (GAČR), grant numbers P202/11/P455, 201/09/P364 and P202/10/0573.

References

- ai Sun, C., Rossing, R., Sinnema, M., Bulanov, P., Aiello, M., 2010. Modeling and managing the variability of web service-based systems. *J. Syst. Software* 83 (3), 502–516.
- Alexe, B., Tan, W.-C., Velegrakis, Y., 2008. STBenchmark: towards a Benchmark for Mapping Systems. *Proc. VLDB Endow.* 1 (1), 230–244.
- Al-Jadir, L., El-Moukaddem, F., 2003. Once upon a time a DTD evolved into another DTD. ... In: *Object-Oriented Information Systems*. Springer, Berlin, Heidelberg, pp. 3–17.
- An, Y., Borgida, A., Mylopoulos, J., 2008a. Discovering and maintaining semantic mappings between XML schemas and ontologies. *J. Comput. Sci. Eng.* 2 (1), 44–73.
- An, Y., Hu, X., Song, I.-Y., 2008b. Round-trip engineering for maintaining conceptual-relational mappings. In: *CAISE '08: Proc. of the 20th Int. Conf. on Advanced Information Systems Engineering*, Springer-Verlag, Berlin, Heidelberg, pp. 296–311.
- Andrikopoulos, V., Benbernou, S., Papazoglou, M.P., 2008. Managing the evolution of service specifications. In: *CAISE '08: Proc. of the 20th Int. Conf. on Advanced Information Systems Engineering*, Springer-Verlag, Berlin, Heidelberg, pp. 359–374.
- Aversano, L., Bruno, M., Penta, M.D., Falanga, A., Scognamiglio, R., 2005. Visualizing the evolution of web services using formal concept analysis. In: *IWPSE '05: 8th Int. Workshop on Principles of Software Evolution*, pp. 57–60.
- Banerjee, J., Kim, W., Kim, H.-J., Korth, H.F., 1987. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Rec.* 16 (3), 311–322.
- Bellahsene, Z., Bonifati, A., Rahm, E., 2011. Schema Matching and Mapping, *Data-Centric Systems and Applications*. Springer, Berlin, Heidelberg, Berlin, Heidelberg.
- Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J., 2007. XQuery 1.0: An XML Query Language. W3C, URL: <http://www.w3.org/TR/xquery/>.
- Boronat, A., Carsí, J.A., Ramos, I., 2006. Algebraic specification of a model transformation engine. In: *FASE '06: Proc. of the 9th Int. Conf. Fundamental Approaches to Software Engineering*, Vienna, Austria, vol. 3922, LNCS, Springer, pp. 262–277.
- Bouchou, B., Duarte, D., Alves, M.H.F., Laurent, D., Musicante, M.A., 2004. Schema evolution for XML: a consistency-preserving approach. In: *Mathematical Foundations of Computer Science*, Springer-Verlag, Prague, Czech Republic, pp. 876–888.
- Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F., 2008. Extensible Markup Language (XML) 1.0, fifth ed. W3C, URL: <http://www.w3.org/TR/2008/REC-xml-20081126/>.
- Cavaleri, F., 2010. EXup: an Engine for the Evolution of XML Schemas and Associated Documents. In: *EDBT '10: Proc. of the 2010 EDBT/ICDT Workshops*, ACM, New York, NY, USA, pp. 1–10.
- Chamberlin, D., Florescu, D., Melton, J., Robie, J., Siméon, J., 2007. XQuery Update Facility 1.0. W3C, URL: <http://www.w3.org/TR/xquery-update-10/>.
- Chen, P., 2002. Entity-relationship modeling: historical events future trends, and lessons learned. In: *Software Pioneers: Contributions to Software Engineering*, Springer, New York, NY, USA, pp. 296–310.
- Cicchetti, A., Ruscio, D.D., Pierantonio, A., 2009. Managing dependent changes in coupled evolution. In: *Proc. of the 2nd Int. Conf. on Model Transformations, ICMT 2009*, Zurich, Switzerland, vol. 5563, LNCS, Springer, pp. 35–51.
- Coox, S.V., 2003. Axiomatization of the evolution of XML database schema. *Program. Comput. Softw.* 29 (3), 140–146.
- Curino, C.A., Moon, H.J., Zaniolo, C., 2008. Graceful database schema evolution: the PRISM workbench. *Proc. VLDB Endow.* 1 (1), 761–772.
- Curino, C., Moon, H.J., Zaniolo, C., 2009. Automating database schema evolution in information system upgrades. In: *HotSWUp '09: Proc. of the 2nd Int. Workshop on Hot Topics in Software Upgrades*, ACM, New York, NY, USA, pp. 1–5.
- Czarnecki, K., Helsen, S., 2006. Feature-based survey of model transformation approaches. *IBM Syst. J.* 45 (3), 621–645.
- Dominguez, E., Lloret, J., Rubio, A.L., Zapata, M.A., 2005. Evolving XML schemas and documents using UML class diagrams. In: *DEXA '05: Proc. of the 16th Int. Conf. on Database and Expert Systems Applications*, vol. 3588, LNCS, Springer, pp. 343–352.
- France, R., Rumpe, B., 2007. Model-Driven Development of complex software: a research roadmap. In: *FOSE '07: 2007 Future of Software Engineering*, IEEE Computer Society, Washington, DC, USA, pp. 37–54.
- Geneves, P., Layaida, N., Quint, V., 2009. Identifying query incompatibilities with evolving XML schemas. In: *ICFP '09: Proc. of the 14th ACM SIGPLAN Int. Conf. on Functional Programming*, ACM, New York, NY, USA, pp. 221–230.
- Hartung, M., Terwilliger, J., Rahm, E., 2011. Recent advances in schema and ontology evolution. In: *Bellahsene, Z., Bonifati, A., Rahm, E. (Eds.), Schema Matching and Mapping, Data-Centric Systems and Applications*. Springer, Berlin, Heidelberg, pp. 149–190 (doi:10.1007/978-3-642-16518-4-6).
- ISO/IEC 9075-14:2003 Part 14: XML-Related Specifications (SQL/XML), Int. Organization for Standardization, 2006.
- Jelliffe, R., 2001. The Schematron – An XML Structure Validation Language using Patterns in Trees, ISO/IEC 19757. URL: <http://xml.ascc.net/resource/schematron/>.
- Klímek, J., Nečáský, M., 2010. Integrating XML schemas for evolution of web services. In: *ICWS 2010: Proc. of The 8th Int. Conf. on Web Services*, IEEE Computer Society, Miami, FL, USA, pp. 307–314.
- Klímek, J., Malý, J., Nečáský, M., 2011. eXolutio – A Tool for XML Data Evolution. URL: <http://exolutio.com>.
- Klettke, M., 2007. Conceptual XML schema evolution – the CoDEX approach for design and redesign. In: *BTW '07, Aachen*, Germany, pp. 53–63.
- Langlois, B., Exertier, D., Bonnet, S., 2006. Performance improvement of MDD tools. In: *EDOCW '06: Proc. of the 10th IEEE on Int. Enterprise Distributed Object Computing Conf. Workshops*, IEEE Computer Society, Washington, DC, USA, p. 19.
- Lerner, B.S., 2000. A model for compound type changes encountered in schema evolution. *ACM Trans. Database Syst.* 25 (1), 83–127.
- Malý, J., Mlýnková, I., Nečáský, M., 2011. XML data transformations as schema evolves. In: *ADBI '11: Proc. of the 15th Advances in Databases and Information Systems*, Springer-Verlag, Vienna, Austria.
- Mens, T., Van Gorp, P., 2006. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.* 152, 125–142.
- Miller, J., Mukerji, J., 2003. MDA Guide Version 1.0.1, Object Management Group. URL: <http://www.omg.org/docs/omg/03-06-01.pdf>.
- Mlýnková, I., 2009. Adaptive XML-to-relational storage strategies. In: *Handbook of Research on Innovations in Database Technologies and Applications: Current and Future Trends*. Idea Group Publishing, pp. 852–859 (February).
- Moro, M.M., Malaika, S., Lim, L., 2007. Preserving XML queries during schema evolution. In: *WWW '07: Proc. of the 16th Int. Conf. on World Wide Web*, ACM, New York, NY, USA, pp. 1341–1342.
- Murata, M., Lee, D., Mani, M., Kawaguchi, K., 2005. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Tech.* 5 (4), 660–704.
- Murata, M., 2002. RELAX (Regular Language Description for XML), ISO/IEC DTR 22250-1. URL: <http://www.xml.gr.jp/relax/>.
- Murzek, M., Kramler, G., Michlmayr, E., 2006. Structural patterns for the transformation of business process models. In: *EDOCW '06: Proc. of the 10th IEEE on Int. Enterprise Distributed Object Computing Conf. Workshops*, IEEE Computer Society, Washington, DC, USA, p. 18.
- Nečáský, M., Mlýnková, I., 2009a. On different perspectives of XML schema evolution. In: *FlexDBIST '09: Proc. of the 5th Int. Workshop on Flexible Database and Information System Technology*, IEEE Computer Society, Linz, Austria.
- Nečáský, M., Mlýnková, I., 2009b. Five-level multi-application schema evolution. In: *DATESO '09: Proc. of the Databases, Texts, Specifications, and Objects*, Matfyz Press, April, pp. 213–217.
- Nečáský, M., Mlýnková, I., 2010. When conceptual model meets grammar: a formal approach to semi-structured data modeling. In: *Chen, L., Triantafillou, P., Suel, T. (Eds.), WISE '10: Web Information Systems Engineering*, vol. 6488, LNCS, Springer, Berlin, Heidelberg, pp. 279–293.
- Nečáský, M., Mlýnková, I., 2010. A framework for efficient design maintaining, and evolution of a system of XML Applications. In: *DATESO '10: Proc. of the Databases, Texts, Specifications, and Objects*, MatfyzPress, April, pp. 38–49.
- Nečáský, M., Opočenská, K., 2009. Designing and maintaining XML integrity constraints. In: *MoViX '09: Proc. of the 1st Int. Workshop on Modelling and Visualization of XML and Semantic Web Data*, IEEE Computer Society, Linz, Austria.
- Nečáský, M., Klímek, J., Kopenc, L., Kučerová, L., Malý, J., Opočenská, K., 2008. XCase – A Tool for XML Data Modeling. URL: <http://xcase.codeplex.com>.
- Nečáský, M., 2009. Conceptual modeling for XML. *Dissertations in Database and Information Systems*, vol. 99. IOS Press, Amsterdam, The Netherlands.
- Nguyen, H.-Q., Taniar, D., Rahayu, J.W., Nguyen, K., 2011. Double-layered schema integration of heterogeneous XML sources. *J. Syst. Software* 84 (1), 63–76.
- Noy, N.F., Klein, M., 2004. Ontology evolution: not the same as schema evolution. *Knowl. Inf. Syst.* 6 (4), 428–440.

- Object Management Group, 2007a. UML Infrastructure Specification 2.1.2 (Nov 2007). URL: <http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/>.
- Object Management Group, 2007b. UML Superstructure Specification 2.1.2 (Nov 2007). URL: <http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/>.
- Object Constraint Language Specification, version 2.0, OMG, 2009. URL: <http://www.omg.org/technology/documents/formal/ocl.htm>.
- OMG, 2005. MOF QVT Final Adopted Specification, Object Modeling Group (June 2005). URL: <http://fparreiras/papers/mof.qvt.final.pdf>.
- Opočenská, K., Kopecký, M., 2008. IncoX – a Language for XML Integrity Constraints Description. In: DATESO '08: Proc. of the Databases, Texts, Specifications, and Objects, pp. 1–12. <http://CEUR-WS.org>.
- Park, C.-S., Park, S., 2008. Efficient execution of composite web services exchanging intensional data. *Inform. Sci.* 178 (2), 317–339.
- Passi, K., Morgan, D., Madria, S., 2009. Maintaining integrated XML schema. In: IDEAS '09: Proc. of the 2009 Int. Database Engineering, Applications Symp., ACM, New York, NY, USA, pp. 267–274.
- Ravichandar, R., Narendra, N.C., Ponnalagu, K., Gangopadhyay, D., 2008. Morpheus: semantics-based incremental change propagation in SOA-based solutions. *IEEE Int. Conf. on Services Computing 1*, 193–201.
- Ryu, S.H., Casati, F., Skogsrud, H., Benatallah, B., Saint-Paul, R., 2008. Supporting the dynamic evolution of web service protocols in Service-Oriented Architectures. *ACM Trans. Web 2 (2)*, 1–46.
- Simanovsky, A.A., 2008. Data schema evolution support in XML-relational database systems. *Program. Comput. Softw.* 34 (1), 16–26.
- Sindhgatta, R., Sengupta, B., 2009. An extensible framework for tracing model evolution in SOA solution design. In: OOPSLA '09: Proc. of the 24th ACM SIGPLAN Conf. Companion on Object Oriented Programming Systems Languages and Applications, ACM, New York, NY, USA, pp. 647–658.
- Stevens, P., 2010. Bidirectional model transformations in QVT: semantic issues and open questions. *Soft. Syst. Model.* 9 (1), 7–20.
- Tan, M., Goh, A., 2005. Keeping pace with evolving XML-based specifications. In: *EDBT '04 Workshops*, Springer, Berlin, Heidelberg, pp. 280–288.
- Tekli, J., Chbeir, R., Yetongnon, K., 2009. Extensible user-based XML grammar matching. In: *Proc. of the 28th Int. Conf. on Conceptual Modeling, ER '09*, Springer-Verlag, Berlin, Heidelberg, pp. 294–314.
- Thompson, H.S., Beech, D., Maloney, M., Mendelsohn, N., 2004. XML Schema Part 1: Structures, second ed. W3C. URL: <http://www.w3.org/TR/xmlschema-1/>.
- Wojnar, A., Mlýnková, I., Dokulil, J., 2010. Structural and semantic aspects of similarity of document type definitions and XML schemas. *Inform. Sci.* 180 (10), 1817–1836 (Special Issue on Intelligent Distributed Information Systems).
- Web Services Business Process Execution Language (WSBP) TC, OASIS, 2007. URL: http://www.oasis-open.org/committees/tc_home.php%3Fwg_abbrev=wsbp.
- Xiong, Y., Hu, Z., Zhao, H., Song, H., Takeichi, M., Mei, H., 2009. Supporting automatic model inconsistency fixing. In: *ESEC/FSE '09: Proc. of the 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp.*, ACM, New York, NY, USA, pp. 315–324.
- MOF 2.0/XML Mapping Specification, v2.1.1, OMG, 2009. URL: <http://www.omg.org/technology/documents/formal/xmi.htm>.
- Yu, C., Popa, L., 2005. Semantic adaptation of schema mappings when schemas evolve. In: *VLDB '05: Proc. of the 31st Int. Conf. on Very Large Data Bases, VLDB Endowment*, pp. 1006–1017.



Martin Nečaský received his Ph.D. degree in Computer Science in 2008 from the Charles University in Prague, Czech Republic, where he currently works at the Department of Software Engineering as an assistant professor. He is an external member of the Department of Computer Science and Engineering of the Faculty of Electrical Engineering, Czech Technical University in Prague. His research areas involve XML data design, integration and evolution. He is an organizer/PC chair/member of more than 10 international events. He has published more than 30 papers (two received Best Paper Award). He has published 3 book chapters and a book.



Jakub Klímeček received his Master's degree in Computer Science in September 2009 from the Charles University in Prague, Czech Republic, where he currently is a Ph.D. student at the Department of Software Engineering. His research areas involve XML data design, integration and evolution. He has published 12 refereed conference papers. He is a co-organizer of 1 local workshop.



Jakub Malý received his Master's degree in Computer Science in June 2010 from the Charles University in Prague, Czech Republic, where he currently is a Ph.D. student at the Department of Software Engineering. His research areas involve conceptual modeling of XML data and evolution of XML applications. He has published 7 refereed conference papers.



Irena Mlýnková received her Ph.D. degree in Computer Science in 2007 from the Charles University in Prague, Czech Republic. She is an assistant professor there and an external member of the Department of Computer Science and Engineering of the Czech Technical University. She has published more than 40 publications (16 recorded in WoS), 4 gained the Best Paper Awards. Her research areas involve management of XML data, structural similarity, analysis of real-world data, synthesis of XML data, XML benchmarking, XML schema inference and application evolution. She is a PC member/reviewer of 14 international events and co-organizer of 3 international workshops.

Chapter 3

Efficient Adaptation of XML Data Using a Conceptual Model

Jakub Malý
Martin Nečaský
Irena Mlýnková

Published in the *International Journal on Information Systems Frontiers*.
Springer Science+Business Media, 2012. ISSN 1387-3326. (in press)

Impact Factor: 0.912
5-Year Impact Factor: 1.074



Efficient adaptation of XML data using a conceptual model

Jakub Malý · Martin Nečaský · Irena Mlýnková

© Springer Science+Business Media, LLC 2012

Abstract One of the prominent characteristics of XML applications is their dynamic nature. Changes in user requirements cause changes in schemas used in the systems and changes in the schemas subsequently make existing documents invalid. In this work, we study two tightly coupled problems—*schema evolution* and *document adaptation*. The presented approach extends an existing conceptual model for evolution of XML applications towards document adaptation, by introducing a formal framework for detecting changes between two versions of a schema. From the detected changes it is possible to create a script that transforms documents valid against the old version of the schema to documents valid against its new version.

Keywords XML schema evolution · Document adaptation · Change management · Conceptual model

This work was supported in part by the Czech Science Foundation (GAČR), grants number P202/10/0573 and P202/11/P455.

J. Malý (✉) · M. Nečaský · I. Mlýnková
Department of Software Engineering, Charles University in Prague, Malostranské nám. 25,
118 00 Praha 1, Czech Republic
e-mail: maly@ksi.mff.cuni.cz

M. Nečaský
e-mail: necasky@ksi.mff.cuni.cz

I. Mlýnková
e-mail: mlynkova@ksi.mff.cuni.cz

1 Introduction

The eXtensible Markup Language (XML) (Bray et al. 2008) is currently a de-facto standard for data exchange, gaining popularity as storage technology and, together with the accompanying technologies, such as XML Schema Definition (XSD) (Thompson et al. 2004; Biron 2004), XPath (W3C 2010a), XQuery (W3C 2010b), XSLT (Kay 2007) etc., it becomes a powerful tool.

Consequently, the amount and complexity of software systems that utilize XML and/or selected XML-based standards and technologies for information exchange and storage grows very fast. The systems represent information in a form of XML documents. One of the crucial parts of such systems are *XML formats* which describe the allowed structure of XML documents. Usually, a system does not use only a single XML format, but a set of different XML formats, each in a particular logical execution part. The XML formats usually represent particular views on the application domain of the software system. For example, a software system for customer relationship management (CRM) exploits different XML formats for purchase orders, customer details, product catalogues, etc. All these XML formats represent different views on the CRM domain. We can, therefore, speak about a *family of XML formats* utilized in by a software system.

Having a system which exploits a family of XML formats, we face the problem of *XML format evolution* as a specific part of evolution of the software system as a whole. The XML formats may need to be evolved whenever user requirements or surrounding environment changes. In our previous work Nečaský et al. (2011a, b), we have introduced a framework for such

evolution of a family of XML formats. The framework considers different levels of abstraction of the XML formats, e.g., conceptual schemas, logical XML schemas or instance XML documents which are mutually interrelated by mappings, which allow for correct evolution. We solved a problem of coherent evolution of XML formats according to changing requirements. This means that when a new requirement appears, it is implemented in the XML formats so that they are updated coherently with each other. However, we have not addressed yet another important part of the problem—adaptation of underlying XML documents when their XML schemas evolve.

Contributions In this paper we focus on the impact of evolution of schemas of XML formats on their instance XML documents. We extend our previously published framework with versioning features and an algorithm for propagation of changes to XML documents—i.e. an algorithm for *XML document adaptation*. Our main contributions can be summed up as follows:

- We propose an extension of our previously published framework for evolution of XML formats with versioning features.
- We provide an algorithm for automatic detection of changes between any two versions of an XML schema and propagation of changes to instance XML documents (in a form of automatically generated version transformation script).
- The XML schemas of XML formats are edited by designers at a more user-friendly level of conceptual schemas instead of technical XML schemas.
- The conceptual schemas enable the user to work also with semantics of the XML formats, not only syntax.
- The framework can decide automatically whether transition from one version of the schema to another requires transformation of the existing documents to make them valid against the new version.

Outline The rest of this paper is structured as follows: Section 1 provides examples of situations where an evolution framework can be used and Section 5 analyzes the related work. Section 1 introduces our conceptual model for XML schema modeling. In Section 1.2 we extend the conceptual model to support evolution and versioning of schemas. Section 2 formally defines types of changes that can occur between two versions of a schema. For each type of change Section 3 describes how documents are adapted using our approach. In Section 4 we discuss the remaining open issues and possible ways to solve them. In Section 6 we conclude.

Relation to previous papers In this paper we continue in our effort towards a robust XML evolution framework which would enable evolution and change management of a set of related XML schemas using a common conceptual model. The conceptual model (without evolution features) was firstly proposed in Nečaský (2009) and later generalized in Nečaský and Mlýnková (2010). In Nečaský and Mlýnková (2009b) we proposed the five-level XML evolution framework and provided a sample set of operations and propagation between the highest two levels—platform-independent and platform-specific. The levels and their mutual relations were formally defined in Nečaský et al. (2011b), the edit operations at the conceptual levels and their propagation to neighboring levels in Nečaský et al. (2011a). Its experimental implementation of the framework can be found at Klímek et al. (2011).

In this paper we focus on the problem of propagating changes from the platform-specific to the extensional (i.e. data) level. We have already briefly published basic ideas of our approach in Malý et al. (2011). In this paper, we provide a detailed and formal description of the solution including all algorithms in the context of our whole evolution framework.

As a motivation for our approach we provide three different scenarios where it can be successfully applied.

Document update after new version adoption In the first scenario, we consider an XML application storing data in XML documents. The documents can be stored in a file system, in an XML-enabled relational database shredded into tables (Oracle XML DB Home. <http://www.oracle.com/technetwork/database/features/xmldb/index.html>) or in a native XML database (eXist <http://www.exist-db.org/>, MarkLogic Server. <http://www.marklogic.com/>). As requirements change, the system designer needs to adjust the XML schemas existing in the system. To keep the system consistent, the documents stored in the system must be transformed so that they are valid against the new version of the schemas. The process of propagation of changes from schemas to documents is called *document adaptation*.

The system designer may choose to adapt the documents manually (i.e. by editing them individually), but the amount of work will grow with the number of documents and the whole process will be time-consuming and error-prone. Alternatively, the user prepares an *adaptation script*—a sequence of commands that can be executed upon all the documents attached to the schema and adapt them all in one batch. Creating such a script from scratch can be difficult and requires a good knowledge of a suitable implementation language. Our approach aims to eliminate these obstacles and reduces

the designer's work to the necessary minimum by generating the adaptation script semi-automatically and using an abstract model rather than working with an implementation language directly.

In case where XML documents are stored as files or in a native database, the adaptation script can be executed upon them directly. When the documents are stored in a relational database, the database vendor provides an interface (Oracle XML DB Developer's Guide—XML Schema Evolution. http://download-uk.oracle.com/docs/cd/B28359_01/appdev.111/b28369/xdb07evo.htm#BCGFEEBB) using which transition to the new version is performed—this interface requires the evolved schema and the adaptation script. Using our approach, the user only needs to evolve the schema, the adaptation script is generated for him/her.

Translation/mediator In the second scenario there are several systems exploiting the same family of XML schemas to communicate with each other. The XML schemas are administered by one of the parties or by a standardization authority which issues the new versions of the standard.

When the new version of the standard is issued, the involved parties are required to adopt the new version. However, some of the parties do not adopt the new version immediately, hence, necessarily, after a certain period of time, there are several versions of the standard deployed and actively used at the same time. This effectively means that the system needs to be able to process different versions of documents. It can be achieved in two ways—either (a) by accepting different versions of the documents on the input and processing each version differently, or (b) by transforming the documents to the latest version before they are processed. A similar situation and solutions are with the output documents (responses)—when a system sends a document valid against a schema which is a part of version v of the standard, it probably expects a result to be also valid against a schema which is a part of v .

The significant benefit of (b) approach is that the business logic of the processing of the input document is compact (and the problem of different versions is solved by a stand-alone component, sometimes called *mediator*), whereas the (a) approach may obfuscate the business logic by introducing new branches, corrections and error recovery sections in the code with each new version supported.

Our framework significantly reduces the effort required when the first way of dealing with different versions of documents on the input/output is selected. It can generate an adaptation script for any two versions of the schema and this adaptation script can be used

to pre-process the documents sent by the parties that have not adopted the latest versions yet in the mediator (and the process is transparent for the business logic components). The only requirement that the designer needs to evolve the old version to the new version in our framework which keeps track between both versions.

Mapping between schemas and system integration The third scenario does not deal with schema evolution in the system, but aims at reducing the effort for integration of schemas concerning the same problem domain. For one business area or problem domain, several independent solutions may emerge. The result is a set of parties using their proprietary schemas. After some time, the involved parties come to the point where they need to interact with each other (they share the same business domain after all), companies may be merged etc. This situation requires *system integration*. One approach is to pick one of the existing solutions or create a new one and unify the participants under this chosen solution. However, this may turn out to be too costly and the participants may instead decide to continue using their proprietary systems and only provide separate interfaces for the other parties. The inter-party communication can then be solved by mappings between the proprietary systems.

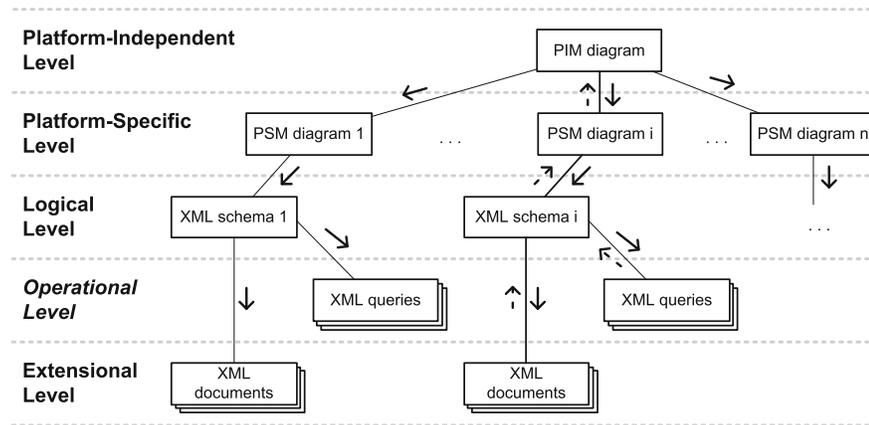
In Klímek and Nečáský (2010), we describe, how our framework deals with the integration problem and helps the user to define mappings between systems. The document adaptation algorithm can utilize these mappings to create an adaptation script and again supply the systems with mediator components (and use generated adaptation scripts in these), so that they will be able to communicate with the other systems using different set of schemas.

Our framework for design and evolution of a family of XML formats applied in a software systems has five separate hierarchical levels and we therefore call it *five-level evolution framework*. The basic structure of the framework is depicted in Fig. 1.

The framework is partitioned both horizontally and vertically. Vertical partitions represent individual XML formats. Horizontal partitions represent different levels which characterize XML formats from different viewpoints. Briefly, for each XML format the levels are the following:

- The *extensional level* contains XML documents formatted according to the XML format. At this level, the XML format is characterized by its instances.
- The *operational level* contains operations performed with the XML documents. These can be

Fig. 1 Five-level XML evolution framework



queries over the instances or transformations of the instances. At this level, the XML format is characterized by operations.

- The *logical level* contains a logical XML schema which specifies the syntax of the XML format. It is expressed in an XML schema language.
- The *platform-specific level* contains a schema which specifies the semantics of the XML format in terms of the level above.
- The *platform-independent level* contains a conceptual schema which describes the information model of the system and covers the semantics of all XML formats in the family in a uniform way. It is common for all XML formats in the family.

As we can see, the framework covers the syntax and semantics of the XML formats as well as their instances and operations performed over the instances. However, the XML documents, queries and schemas at different horizontal levels are not the only first-class citizens of our framework. There are also mappings between the horizontal levels. They are depicted as lines connecting the levels.

The mappings are crucial for correct evolution of the XML formats. Briefly, evolution means that whenever a change at any place of the framework is performed by a user, the change is propagated to all other affected parts. The need for change propagation is invoked by the mappings. The propagation ensures that the affected parts are adapted so that their consistency with the changed part and with each other is preserved.

1.1 Framework horizontal levels

Let us now describe the horizontal levels in a more detail.

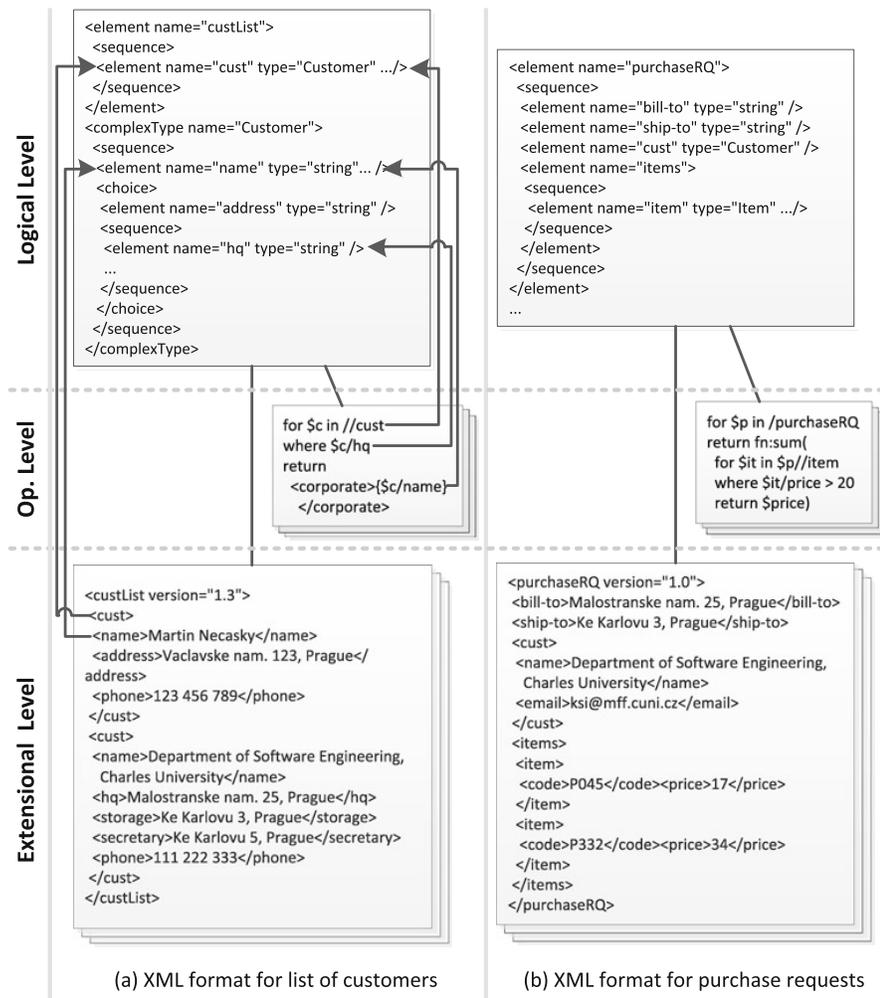
1.1.1 Logical, operational and extensional level

The lowest level, called *extensional level*, represents particular XML schema instances that occur in the system. The instances are XML documents which are persistently stored in an XML database or exchanged between parts of the system or between the system and other systems as messages. The level one step higher, called *operational level*, represents operations over the instances. These might be, e.g., XML queries over the instances expressed in XQuery or transformations of the instances expressed in XSLT. The level above, called *logical level*, represents logical schemas that describe the structure of the instances. They are expressed in a selected XML schema language, e.g. DTD (Bray et al. 2008), XSD, RELAX NG (Clark and Makoto 2001), Schematron (ISO 2005), etc. We demonstrate the three levels in Fig. 2. It shows our two sample XML formats represented at the three levels.

There are two kinds of mappings depicted between the three levels. There are mappings of instance XML documents to their XML schemas. The instances are XML documents *valid* against the XML schema. An instance XML element or attribute is mapped to its respective definition in the XML schema. The mapping is created automatically during XML document validation. For example, XML elements *cust* in the instances of XML format on the left of Fig. 2 are mapped to the definition of the XML element *cust* in the XML schema. A valid instance is fully mapped to its XML schema.

The other kind are mappings of operations to XML schemas. Operations are based on the XPath language whose basic construct is a *path* comprising steps which select XML elements and attributes from the instance XML documents. The steps also specify required hierarchical relationships between the selected XML

Fig. 2 Two sample XML formats represented in the framework



elements and attributes (e.g. parent/child or ancestor/descendant). A path is mapped to a respective chain of XML element or attribute definitions in the XML schema. The definitions are in the structural relationship specified by the path steps. The mapping is created automatically during the validation of a path (similarly to validation of XML documents). For example, there are the following paths in the query for the XML format on the left of Fig. 2: `//cust/hq` and `//cust/name`. They are mapped to the corresponding XML element definitions as depicted by the arrows. A correct XML query has all its paths mapped.

Even these three levels indicate problems related to XML evolution. A change in one of them can trigger respective changes in the other two levels. Therefore, we need a mechanism which correctly propagates a change to the other levels. When the structure of an

XML schema changes, its instances and related queries must be adapted accordingly so that their validity and correctness is preserved respectively. Some changes can be propagated automatically, in some cases user interaction is necessary.

1.1.2 Platform-independent and platform-specific levels

If we consider only the three described levels we have no explicit relationship between the vertical partitions, i.e. between the XML formats applied in the system. As we have already discussed, a change in one XML format can trigger changes in the other XML formats to keep their consistency. Therefore, a change in one XML schema must be propagated to the other affected XML formats manually by a designer. This is, of course, highly time-consuming and error-prone solution. The

designer must be able to identify all the affected formats and propagate the change correctly. Often, (s)he is not able to do such a complex work and needs a help of a domain expert who understands the problem domain, but is, typically, a business expert rather than a technical XML format expert. Therefore, it is very hard for him to navigate in the logical XML schemas, operations and instances.

To overcome these problems, we introduce two additional levels to the framework which represent two additional levels of abstraction of the XML schemas. The levels are motivated by the MDA (Miller and Mukerji 2003) principles. The topmost one is a *platform-independent level* which comprises a single *schema in a platform-independent model (PIM schema)*. The PIM schema is a conceptual schema of the application

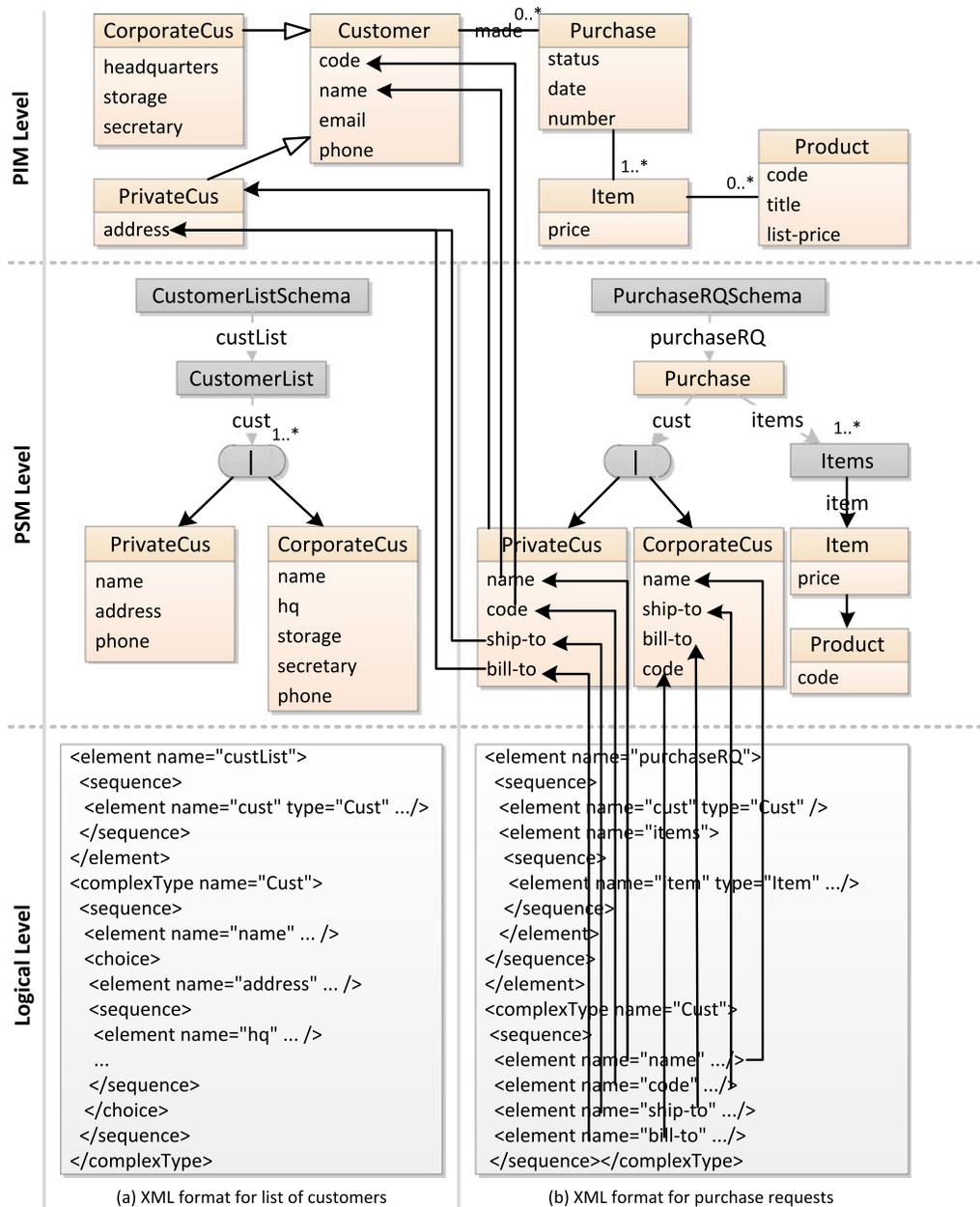


Fig. 3 Two sample XML formats represented at logical, PSM and PIM levels

domain. We use the notation of UML class diagrams to express PIM schemas. A sample PIM schema modeling the domain of customers and their purchases is depicted in Fig. 3.

The level below is a *platform-specific level* which comprises a separate *schema in a platform-specific model (PSM schema)* for each XML format in the family. A PSM schema is also expressed in the notation of UML class diagrams. However, we extended UML notation in Nečaský et al. (2011b) so that it can be used for modeling XML formats. First, a PSM schema has a strictly hierarchical structure. Second, there are few extending modeling constructs. For example, we use so called *choice content model* which is depicted as an oval with the | symbol inside. It models a choice in XML content which is usual in XML formats. Two sample PSM schemas for our two XML formats are depicted in Fig. 3.

A PSM schema models an XML format and can be viewed from two perspectives—*conceptual* and *grammatical*. The *conceptual perspective* models the semantics of the XML format in terms of the conceptual schema from the PIM level. The semantics is expressed as an unambiguous mapping of the components of the PSM schema to the components of the PIM schema. We demonstrate the mapping in Fig. 3 on the right-hand side PSM schema. There is depicted the mapping of PSM class *PrivateCus* to PIM class *PrivateCus* and its PSM attributes *name*, *shipto*, *billto*, *code* to PIM attributes *name*, *address*, and *code*, respectively. The PSM attributes *shipto* and *billto* are mapped to the same PIM attribute. Therefore, the semantics of the portion of the PSM schema is that *PrivateCus* class models a private customer with a name and code. Both shipping and billing address referred to in the purchase are the same address evidenced in the system for the customer. The PSM class *CorporeCus* is mapped similarly but its PSM attributes *shipto* and *billto* are mapped to the PSM attributes *storage* and *headquarters*, respectively. In other words, the semantics of *shipto* and *billto* attributes in the modeled XML formats is different for the private and corporate customers. Associations are mapped as well. For example, both PSM associations going to *PrivateCus* and *CorporeCus* are mapped to the PIM association connecting the PIM classes *Customer* and *Purchase*. There can also be components which are not mapped. They are displayed in the grey color, e.g. class *Items* or PSM associations *cust* and *items* in the PSM schema on the right. These components have no semantics. in the PSM schema on the right. These components have no semantics.

From the grammatical perspective, a PSM schema models a grammar of the respective XML format. In

other words, it models the syntax of the XML format which is expressed at the logical level as an XML schema. The conversion of the PSM schema to a corresponding XML schema is automatic. Briefly, a class models a sequence of XML element and attribute declarations. An association with a name models an XML element whose content is the sequence modeled by its child class. The purpose of the choice content model is to support variants in the XML format. In most cases, the choice content model will actually allow for different XML contents. That is the case of the left-hand side schema in our example models (this case is translated into *choice* construct at the logical level). However, in some cases (as illustrated in the right-hand side schema), the branches are equivalent from the grammatical perspective (but not from the conceptual, as we have shown) and at the logical level joined into one.

During the conversion, an unambiguous mapping of the XML schema components to the PSM schema components is automatically created. We depict a portion of a sample mapping of XML elements *name*, *ship-to*, etc. to PSM attributes of classes *PrivateCus* and *CorporeCus* in Fig. 3.

In Nečaský et al. (2011b) we have formally shown that the expressive power of our PSM schemas is the same as the expressive power of *regular tree grammars (RTG)* (Murata et al. 2005). RTG is a notation which allows to express an XML schema in a formal way and allows for reasoning about an expressive power of various XML schema languages. It has been shown in Murata et al. (2005) that each schema expressed in XSD can be also expressed as RTG.

The result is a hierarchy which interconnects all the XML formats in the family using the common PIM schema. The change propagation between different XML formats is realized using this common point. For instance, if a change occurs in a selected XML document, it is first propagated to the respective XML schema, PSM schema and, finally, to the PIM schema. We speak about an *upwards propagation*, in Fig. 1 represented by the upwards arrows. It enables one to identify the part of the problem domain that was affected. Then, we can invoke the *downwards propagation*. It enables one to propagate the change of the problem domain to all the related parts of the system. In Fig. 1 it is denoted by the downwards arrows.

In our previous work Nečaský et al. (2011a, b) we have introduced the mechanism of propagating changes between PSM and PIM levels in both directions. This also includes the logical level since the PSM level is the full representation of the logical level (see Nečaský et al. 2011b). In this paper, we concentrate on

propagating changes from an XML schema expressed at the PSM level to the extensional level. Propagating changes from the extensional level to the logical level and considering the operation level is our future work. We have already developed an algorithm for propagating changes from the extensional to the logical level and implemented it in an experimental tool called *jInfer*.¹ The algorithm represents an XML schema as a regular tree grammar (Nečaský et al. 2011b). It takes a set of XML documents as an input and recognizes a grammar describing their structure. It can also consider the old version of the XML schema such that the XML documents were valid against it but were modified so that they are no longer valid. The algorithm adapts the XML schema in a minimal way so that the XML documents are valid again. However, the full description of the algorithms in the context of our evolution framework is still our future work.

1.2 Selected part of the problem

In this work, we are interested in one particular problem in the context of the introduced framework which is adaptation of instance XML documents of a single XML format when the XML schema of the format is changed. As the framework shows, we can consider changes to the XML format at the two levels—logical or PSM level. The former one specifies the syntax of the XML format while the latter one models both syntax and semantics in terms of the PIM schema.

Working at the logical level means that an XML schema is evolved. It enables us to adapt to structural changes, but we are not able to identify changes in the semantics. Consider the example provided by authors of *CoDEX* (Klettke 2007),² where the user moves element `address` from element `owner` to element `producer`. The structure of both the elements `address` is the same (they would probably refer to the same type), but semantically the meaning is different (the address of the producer is not the same as the address of the owner). This evolution step is depicted in Fig. 4. Both the old (a) (before `address` was moved) and new version (b) have the same PIM model (the PIM model was not evolved), at other levels, they differ. When class `Address` is moved from `Owner` to `Producer`, it can be observed, that the structure remains the same (at the logical and operational levels); however, the PSM schema is

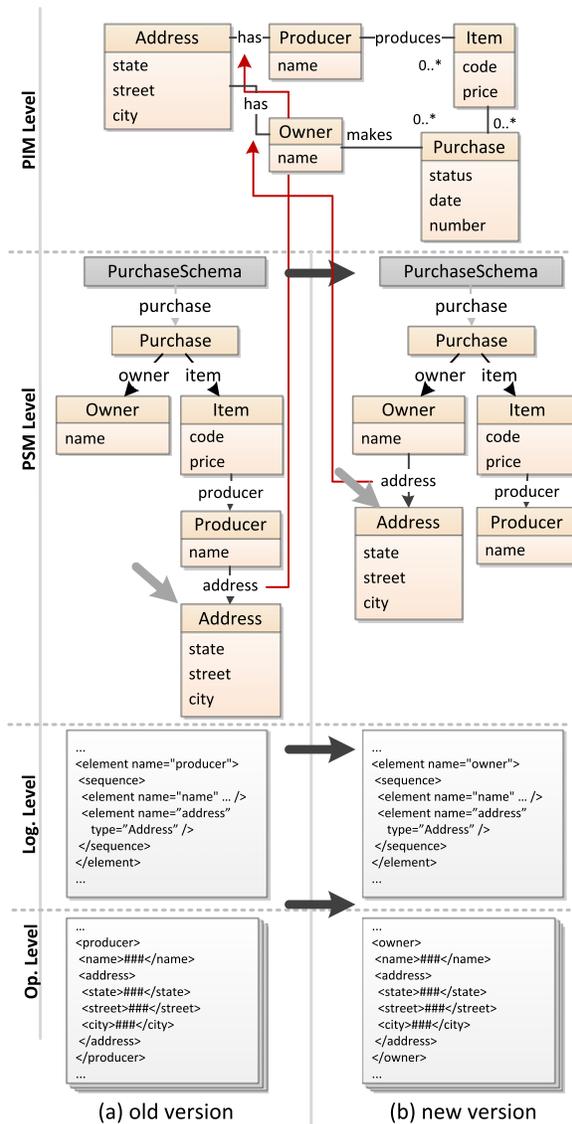


Fig. 4 A sample XML format in two versions

different—the association leading to class `Address` is mapped to a different PIM association, than in the old version (`Producer-Address` vs. `Owner-Address`). Propagating this change by moving the contents of `address` from element `owner` to element `producer` would create valid, yet semantically incorrect document. With the two-layer conceptual model, the system can identify that `owner` is a different concept than `producer` and will not try to adapt the documents by moving `address` element, but by a combination of deleting and inserting. If we work at the PSM level, we are also able to identify changes in the semantics. In

¹<http://jinfer.sourceforge.net/index.html>

²More on this system can be found in Section 5

that case we are able to easily recognize the above mentioned change and adapt the XML documents correctly.

Working at the PSM level (PSM schemas) instead of at the logical (XML schemas, e.g. XSDs) level has additional benefits from the user point of view. PSM schemas are much more transparent and easier to read and work with than lengthy XSDs. The same holds for examining the list of changes between the old and evolved PSM schemas compared to list of changes between two XSDs. Also, since PSM schema has less types of constructs than XML Schema language, the comparison algorithm will be less complex.

Without any loss of generality, we therefore consider that the XML format is changed at the PSM level, i.e. that the designer changes the PSM schema of the evolved XML format. We extend our framework with a formalism for versioning of PIM and PSM schemas. We further study the impact of changes in a PSM schema to its instance XML documents and introduce a method which correctly adapts them so that their validity is restored.

First, we introduce the formal model of PIM and PSM schemas (Nečaský et al. 2011b) and extend it with a completely new formalism for their versioning. The formalism allows for modeling more versions of PIM and PSM schemas in parallel. We also show how concepts in different versions are connected and why we need such connections for document adaptation.

1.3 Formal definitions of PIM and PSM

Even though both PIM and PSM schemas are basically UML class diagrams, we introduce our own formal definitions of PIM and PSM schemas. This is necessary because the original specification of UML class diagrams was not formal enough for our purposes.

Let us start with the PIM schema. As we have mentioned, it is a conceptual schema of the problem domain based on the classical model of UML class diagrams (Object Management Group 2007a, b). For simplicity, we use only its basic constructs: classes, attributes and binary associations.

Definition 1 A *platform-independent schema (PIM schema)* is a triple $\mathcal{S} = (\mathcal{S}_c, \mathcal{S}_a, \mathcal{S}_r)$ of disjoint sets of *classes*, *attributes*, and *associations*, respectively.

- A *Class* $C \in \mathcal{S}_c$ has a name assigned by function *name*.
- An *Attribute* $A \in \mathcal{S}_a$ has a name, data type and cardinality assigned by functions *name*, *type*, and *card*, respectively. Moreover, A is associated with a class from \mathcal{S}_c by function *class*.

- An *Association* $R \in \mathcal{S}_r$ is a set $R = \{E_1, E_2\}$, where E_1 and E_2 are called *association ends* of R . R has a name assigned by function *name*. Both E_1 and E_2 have a cardinality assigned by function *card* and are associated with a class from \mathcal{S}_c by function *participant*. We will call *participant*(E_1) and *participant*(E_2) *participants* of R . *name*(R) may be undefined, denoted by *name*(R) = λ .

For a class $C \in \mathcal{S}_c$, *attributes*(C) denotes the set of attributes of C and *associations*(C) denotes the set of associations with C as a participant.

PIM schema components have usual semantics: a class models a real-world concept, an attribute of that class models a property of the concept, and, an association models a kind of relationships between two concepts modeled by the connected classes. Note that an association is formally an unordered set of its two endpoints. In other words, associations are unordered in PIM schemas. Even though UML allows for directed associations we do not consider them in this paper. For our purposes, it is not important whether an association is ordered or not at the PIM level. At the PSM we handle ordered and unordered associations from the PIM level in the same way. A sample PIM schema is depicted in Fig. 5.

A PSM schema represents a part of the PIM schema and extends it with details of its representation in a particular XML format. We model PSM schemas as UML class diagrams extended for the purposes of XML schema modeling. The extension is necessary because of several specifics of XML (such as hierarchical structure or distinction between XML elements and attributes) which cannot be modeled by standard UML constructs.

Definition 2 A *platform-specific model schema (PSM schema)* is a 5-tuple $\mathcal{S}' = (\mathcal{S}'_c, \mathcal{S}'_a, \mathcal{S}'_r, \mathcal{S}'_m, \mathcal{C}_{\mathcal{S}'})$ of disjoint sets of *classes*, *attributes*, *associations*, and *content models*, respectively, and one specific class $\mathcal{C}_{\mathcal{S}'} \in \mathcal{S}'_c$ called *schema class*.

- A *Class* $C' \in \mathcal{S}'_c$ has a name assigned by function *name*.
- An *Attribute* $A' \in \mathcal{S}'_a$ has a name, data type, cardinality and XML form assigned by functions *name*, *type*, *card* and *xform*, respectively. *xform*(A') $\in \{e, a\}$. Moreover, it is associated with a class from \mathcal{S}'_c by function *class* and has a position assigned by function *position* within the all attributes associated with *class*(A').
- An *Association* $R' \in \mathcal{S}'_r$ is a pair $R' = (E'_1, E'_2)$, where E'_1 and E'_2 are called *association ends* of

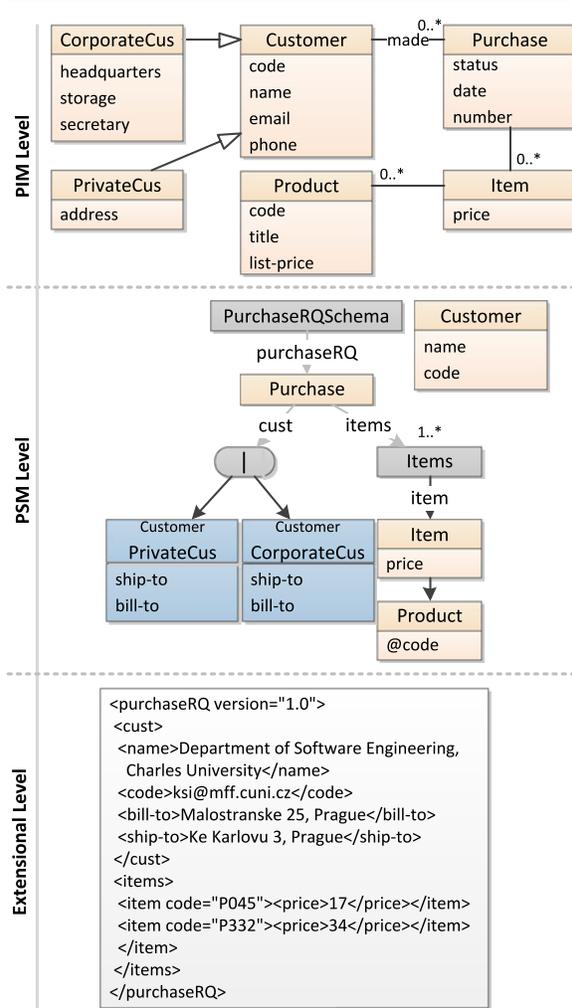


Fig. 5 Sample PIM and PSM Schemas. Conceptual and grammatical perspective of a PSM schema

R' . Both E'_1 and E'_2 have a cardinality assigned by function $card$ and each is associated with a class from S'_c or content model from S'_m assigned by function $participant$, respectively. We will call $participant(E'_1)$ and $participant(E'_2)$ *parent* and *child* and will denote them by $parent(R')$ and $child(R')$, respectively. Moreover, R' has a name assigned by function $name$ and has a position assigned by function $position$ within all the associations with the same $parent(R')$. $name(R')$ may be undefined, denoted by $name(R') = \lambda$.

- A *Content model* $M' \in S'_m$ has a content model type assigned by function $cmtpe$. $cmtpe(M') \in \{\text{sequence, choice, set}\}$.

The graph $(S'_c \cup S'_m, \{(n_1, n_2) : n_1, n_2 \in S'_c \cup S'_m \wedge (\exists(E'_1, E'_2) \in S'_r) (participant(E'_1) = n_1 \wedge participant(E'_2) = n_2)\})$ must be a forest³ of rooted trees with one of its trees rooted in C_S . We will use the notion $N' = S'_c \cup S'_m$ for the nodes of this forest. For a class $C' \in S'_c$, $attributes(C')$ denotes the sequence of attributes of C' ordered by *position* and $content(C')$ denotes the sequence of associations with C' as a parent ordered by *position*. We denote $content(C')$ *content of* C' . Similarly, for a content model $M' \in S'_m$, $content(M')$ denotes the sequence of associations with M' as a parent ordered by *position*.

As we have already discussed in Section 1, we view a PSM schema from two perspectives—*conceptual* and *grammatical*. From the conceptual perspective, a PSM schema models the semantics of an XML format in terms of a PIM schema. This is formally expressed by mapping classes, attributes and associations in the PSM schema to their PIM equivalents. We call the mapping *interpretation of the PSM schema against the PIM schema*. The interpretation cannot be arbitrary. There are some restrictions which prevent from semantic inconsistencies. For example, a PSM attribute cannot be mapped to an arbitrary PIM attribute—the class of the PSM attribute must be mapped to the class of the PIM attribute. Without this condition, it would be possible to map, e.g., PSM attribute $code'$ of PSM class $Customer'$ in our sample PSM schema depicted in Fig. 5 to PIM attribute $number$ of PIM class $Purchase$. Intuitively, this does not make any sense, but our condition prevents from this mapping explicitly. It is forbidden because $class(code') = Customer'$ is mapped to $Customer$ which is not the class $class(number) = Purchase$. In other words, the condition preserves semantic consistency between PSM and PIM attributes.

In a similar fashion, we restrict association mappings. Suppose a PSM association R' . Let $child(R') = D'$ be mapped to PIM class D . Let the closest ancestor class of R' which is mapped to the PIM schema be a class C' . Let C' be mapped to PIM class C . If C' and D' do not exist, R' cannot be mapped at all. Otherwise, it can be mapped only to PIM association R which connects C and D or their inheritance descendants. Similarly to the attribute condition, the association condition preserves semantic consistency between PSM and PIM associations. Without the condition, it would be possible to map, e.g., PSM association R' connecting PSM classes $Items'$ and $Item$ in our sample PSM schema to PIM

³Note that since S' is a forest, we could model R' directly as a pair of connected components. However, we use association ends to unify the formalism of PSM with the formalism of PIM.

association R connecting PIM classes *Purchase* and *Customer* which, intuitively, does not make sense.

There can also be components which are not mapped to the PIM schema. These components are displayed in the gray color. We refer to Nečaský et al. (2011b) for detailed and formal description.

From the grammatical perspective, a PSM schema models the syntax of the XML format. In other words, it models the XML schema of the XML format. For example, the PSM schema depicted in Fig. 5 models the syntax of the XML format whose instance is depicted in the same figure. The PSM schema does not depend on any particular XML schema language. It can be automatically translated to an arbitrary language. In Nečaský et al. (2011b) we showed how a PSM schema can be translated to a regular tree grammar (Murata et al. 2005) which can be expressed in XSD or RELAX NG.

Here, we describe only briefly which XML structures are modeled by PSM components. Let us start with the schema class of a PSM schema. It models whole XML documents. In our example, the schema class, named `PurchaseRQSchema'`, models whole XML documents with purchase requests.

All other classes model a complex content which comprises of a set of XML attributes and of a sequence of XML elements. The set of XML attributes is modeled by the class' attributes (s.t. $xform(A') = e$, see the next paragraph), the sequence of XML elements by the class' attributes and child associations. For example, class `Purchase'` models an XML content represented by its two child associations `cust'` and `items'`. Class `Item'` models an XML content represented by its attribute `price'` and the association going to child class `Product'`.

A class attribute A' models an XML element or XML attribute depending on its XML form. If $xform(A') = e$ then A' models an XML element. Otherwise (when $xform(A') = a$), A' models an XML attribute. The XML element or attribute name is given by $name(A')$. Visually, the XML form is distinguished by @ symbol for attributes with attribute XML form. For example, the attribute `code'` of class `Product'` models an XML attribute `code`. On the other hand, the attribute `price'` of class `Item'` models an XML element `price`.

An association R' models how the complex content modeled by its child is nested in the complex content modeled by the parent. It is therefore directed from the parent to the child. If the name of R' is defined, the complex content modeled by the child is enclosed in an XML element with the name given by the name of R' . For example, the association `cust'` has name `cust`. It

specifies that the complex content modeled by its child is enclosed in the XML element `cust` which is nested in the complex content modeled by the parent class `Purchase`. On the other hand, the association connecting `Item'` and `Product'` does not have a name defined and, therefore, models only the hierarchical structure but no XML element. If the parent of R' is the schema class then R' must have a name defined and models a root XML element. In our sample, this is the case of the association with the child `Purchase'`. It models the root XML element `purchaseRQ` because of its parent association with name `purchaseRQ`.

There are two PSM specific constructs. The first one is called *content model* which was introduced in Definition 2. By default, child associations of a class in a PSM schema model specify a sequence content model. However, sometimes it is necessary to specify a choice or set content model. For this, we use content models in PSM schemas. They are displayed as small ovals with a specific symbol inside “|” for choice, “{” for set and “...” for sequence. Our sample PSM contains a choice content model. In this particular case it specifies that content of an XML element `cust` (modeled by the parent association of the choice content model) is one of the contents modeled by classes `PrivateCus'` and `CorporateCus'`.

The other construct is called *structural representative*. A structural representative is a class which refers to another class (in the same or another PSM schema). The structural representative extends the complex content modeled by the referred class. A structural representative is displayed as a class with the blue background. For example, there are two structural representatives `PrivateCus'` and `CorporateCus'` of class `Customer'` in our sample PSM schema. They model the same complex content as `Customer'` and extend it with their own content. Note that we cannot move the attributes `ship-to'` and `bill-to'` to `Customer'` because they have different semantics (different mapping to the PIM schema) even though they are equivalent from the grammatical perspective.

Definition 3 Let $S' = (S'_c, S'_a, S'_r, S'_m, C_{S'})$ be a PSM schema and C' be a class from S'_c . C' may be a *structural representative* of another class D' in S'_c which is assigned to C' by function $repr$ ($repr(C') = D'$). If $repr(C')$ is undefined, denoted by $repr(C') = \lambda$, we say that C' is not a structural representative of any class. Let $repr^*(\lambda) = \{\}$ and $repr^*(C') = \{repr(C')\} \cup repr^*(repr(C'))$ where $C' \neq \lambda$. It must hold that $C' \neq repr^*(C')$.

In Nečaský et al. (2011b) we have formally shown that the expressive power of our PSM schemas is the

same as the expressive power of regular tree grammars (RTG) (Murata et al. 2005), i.e. each schema S' can be translated into a corresponding regular tree grammar $G_{S'}$ and vice versa. This allows us to introduce the notion of validity of an XML document against a PSM schema.

Definition 4 For a schema S' , the set of conforming documents $\mathcal{T}(S')$ equals to the language $\mathcal{L}(G_{S'})$ generated by a grammar $G_{S'}$. We will say that an XML document T is *valid* against S' if $T \in \mathcal{T}(S') = \mathcal{L}(G_{S'})$.

For the algorithms presented in this paper, we require PSM schemas to fulfill certain additional conditions. A PSM schema fulfilling these conditions will be called *normalized*.

Definition 5 Let S' be a PSM schema. We call S' *normalized PSM schema* when the following conditions are satisfied:

$$(\forall R' \in \text{content}'(C'_S))(name'(R') \neq \lambda \wedge card'(R') = 1..1) \quad (1)$$

$$(\forall R' \in S'_r)(child'(R') \in S'_m \rightarrow name'(R') = \lambda) \quad (2)$$

$$(\forall M' \in S'_m)(\exists R' \in S'_r)(child'(R') = M') \quad (3)$$

$$\forall C' \in S'_c \setminus \{C'_S\}((\exists R' \in S'_r)(child'(R') = C') \rightarrow (\exists C'_0 \in S'_c)(repr'(C'_0) = C')) \quad (4)$$

If S' does not satisfy some of the conditions 1–4, it is called *relaxed PSM schema*.

The main purpose of normalization is to remove redundancies and unreachable portions of the schema. A normalized schema is simpler than its relaxed equivalent. In Nečáský et al. (2011b), we presented, in the form of an algorithm, how every PSM schema S' can be normalized to schema $\overline{S'}$ and proved formally that normalization does not reduce the modeled language (i.e. $\mathcal{L}(G_{S'}) = \mathcal{L}(G_{\overline{S'}})$). In this paper, we exploit these previous results and work only with normalized schemas. Therefore, the set of different types of possible edit operations we need to consider will be smaller (e.g. we do not have to consider an operation for moving a content model to a root) and, therefore, the introduced adaptation algorithms are less complex.

Condition 1 requires that an association with the parent being the schema class C'_S , has a name and that its cardinality is 1..1. This is natural because each such association models a root XML element. Therefore, its name needs to be specified and it has no sense to

specify a cardinality different from 1..1. Condition 2 requires that an association with a content model as a child does not have a name. The names of associations specify element names. However, an association with a content model as a child does not model an element but only a part of the content of an element. Therefore, its name would not be used anyway. Condition 3 requires that each content model is a child of an association. A content model which is a root is unreachable in the schema and, therefore, redundant. The last condition 4 requires that a class which is a root has a structural representative. Otherwise, the class would be unreachable.

1.4 Versions of the model

One of our objectives was to allow the user to evolve schemas and create new versions, but also let him/her work with the old versions as well. In other words, each version must be independent of the others and the old version should not be lost and replaced by a new version. That is why in our framework, the user can choose any existing version v of the model and via the *branch* operation, create a new version \tilde{v} as a copy of v (branch creates copies of all the concepts and their properties). Then, the user can evolve \tilde{v} to a desired state, but also go back to work with v or any other version existing in the system.

We suppose that each version is identified by some label. In real systems, the labels usually are, e.g., “1.0”, “1.1”, “2.0”, “2.1”, etc. We will use \mathcal{V} to denote the set of labels of all versions of the system. We will call the members of \mathcal{V} *versions labels* or simply *versions*. Initially it has one member v_0 which denotes the initial version of the system. A new version is established and added to \mathcal{V} each time the user executes the *branch* operation.

Versions in \mathcal{V} allow us to say, e.g., “class C belongs to version v ”. We will use function *ver* which answers questions like “To which version does class C belong?”. To define the domain of function *ver* (i.e. everything, that can be versioned in our framework), we will use the following auxiliary definition:

Definition 6 Let \mathcal{S} be a PIM schema and S' be a PSM schema. The set of all the components in \mathcal{S} and S' will be denoted $S_{\text{all}} = S_c \cup S_r \cup S_a$ and $S'_{\text{all}} = S'_c \cup S'_r \cup S'_a$, respectively.

Further, let \mathcal{S}^* be a set of PIM schemas and $\mathcal{S}^{*'}$ a set of PSM schemas. We will use \mathcal{M}^* to denote the set of all schemas in \mathcal{S}^* and $\mathcal{S}^{*'}$, and all components of the schemas. I.e. $\mathcal{M}^* = \mathcal{S}^* \cup \mathcal{S}^{*'} \cup (\bigcup_{S \in \mathcal{S}^*} S_{\text{all}}) \cup (\bigcup_{S' \in \mathcal{S}^{*'}} S'_{\text{all}})$.

Function *ver* specifies which version each schema or component belongs to.

Definition 7 The function $ver : \mathcal{M}^* \rightarrow \mathcal{V}$ assigns a version to each PIM schema, PSM schema and to each of their components. In other words,

Values of function *ver* form the input of the change detection algorithm. When this function is implemented in a tool, the values of *ver* are automatically defined as the user edits the schemas and creates new versions (using operation *branch* with usual semantics).

Figure 6 shows two versions of a PIM schema and one PSM schema. The second version \tilde{v} was created from the first version v using *branch* operation. Branch adds a new version to \mathcal{V} (so that $\mathcal{V} = \{v, \tilde{v}\}$) and defines values of *ver* for the branched schemas and constructs. When system is branched, function *ver* would return v for the PIM and PSM schema on the left side and all their constructs. For the schemas on the right side and their constructs, it would return \tilde{v} . Each time a construct x is added to a schema S , $ver(x)$ is set to $ver(S)$. When a new PSM schema S' is mapped to a PIM schema S , $ver(S')$ is set to $ver(S)$. After being branched, the two versions can be edited separately. In the example, the user decided to reconnect PIM association Address–Purchase to Address–Customer,

rename it from *delivered* to *has* and adapt the PSM schema accordingly.

Definition 8 Let \mathcal{S}^* be a set of PIM schemas and \mathcal{S}'^* a set of PSM schemas. Let $\mathcal{K} \subseteq \mathcal{M}^*$ and $v \in \mathcal{V}$. We will use $\mathcal{K}[v] = \{C | ver(C) = v\}$ to denote *projection of K to version v* or simply *version projection*. In other words, version projection $\mathcal{K}[v]$ returns members of \mathcal{K} that belong to version v .

We require the following conditions to hold:

$$(\forall v \in \mathcal{V})(|\mathcal{S}^*[v]| = 1) \tag{5}$$

$$(\forall \mathcal{S} \in \mathcal{S}^*)(\mathcal{S}_{all} \subseteq \mathcal{M}^*[ver(\mathcal{S})]) \tag{6}$$

$$(\forall \mathcal{S}' \in \mathcal{S}'^*)(\mathcal{S}'_{all} \subseteq \mathcal{M}^*[ver(\mathcal{S}')] \tag{7}$$

The conditions in Definition 8 require that exactly one PIM schema exists for each version (Eq. 5) and that all components of a given PIM or PSM schema belong to the same version (Eqs. 6 and 7). We also require consistency in PSM–PIM mappings (*interpretations* of PSM constructs can be only the PIM constructs from the same version).

In our examples, we will usually show the system divided into version projections. Figure 6 shows the two version projections $\mathcal{M}^*[v]$ and $\mathcal{M}^*[\tilde{v}]$, $\mathcal{M}^*[v]$ on blue background, $\mathcal{M}^*[\tilde{v}]$ in orange background.

Without any loss of generality, in the following text we will assume $|\mathcal{V}| = 2$, unless explicitly stated otherwise (i.e. we expect there are two versions in the system—the old version ($v \in \mathcal{V}$) and the new version ($\tilde{v} \in \mathcal{V}$)).

1.5 Document adaptation in a versioned system

Document adaptation is a process triggered by schema evolution. By schema evolution we mean conducting certain operations upon the existing schema until reaching the desired final state—new version of the schema. Such a change can violate validity of instances of the schema, i.e. XML documents. The state which requires adaptation can be defined as follows:

Definition 9 We say that the set of conforming documents $\mathcal{T}(S')$ of schema S' was *invalidated in the new version* (or just *invalidated*) if: $\exists T \in \mathcal{T}(S') : T \notin \mathcal{T}(\tilde{S}')$. If no such T exists, then S' is called *backwards-compatible*.

The goal of adaptation is to modify the XML documents according to changes in the schema so that they are valid against the new version of the schema.

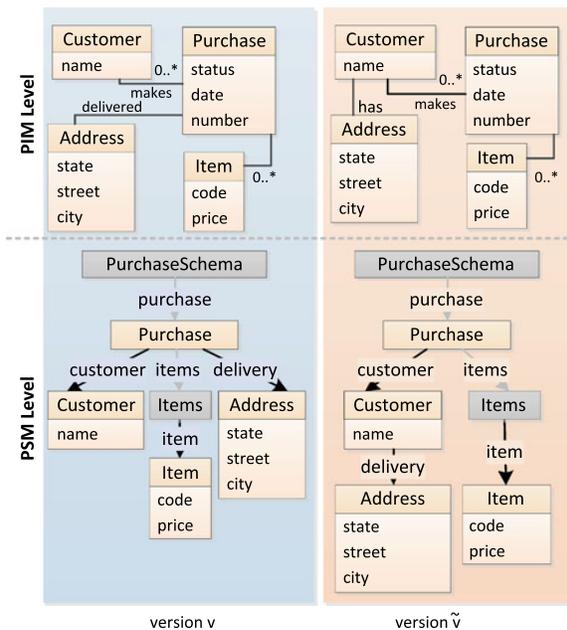


Fig. 6 Two versions of a PIM and a PSM schema

The document adaptation process starts with change detection—two schemas are compared and from the set of detected changes the system deduces steps required for successful document adaptation. Detecting changes in an XML schema or a model of an XML schema is not always straightforward; some differences between the old and new version can be interpreted in more than one way. For example, consider the PSM schemas in Fig. 7.

There are two possible interpretations for evolution of schema in Fig. 7a: either (1) attribute ID was removed and new attribute SSN was added or (2) attribute ID was renamed to SSN. When deciding which interpretation is the correct one, mapping to a PIM schema can be taken into account. E.g. if the attributes are mapped to PIM attributes p_1 and p_2 , whereas p_2 is a new version of p_1 , we can assume that the second interpretation is correct and the attribute was only renamed. But even this is still only a heuristic.

Likewise, there can be two interpretations of the change depicted in Fig. 7b: (1) attribute price was moved from Item to Purchase or (2) the attribute was removed from Item and a new attribute was added to Purchase, having the same name coincidentally. The adaptation of the documents in this example is even more difficult to decide. In the second case, a correct value must be assigned to the new attribute. In the first case, the value of attribute price should be set to the sum of the values of price in all the items of the purchase.

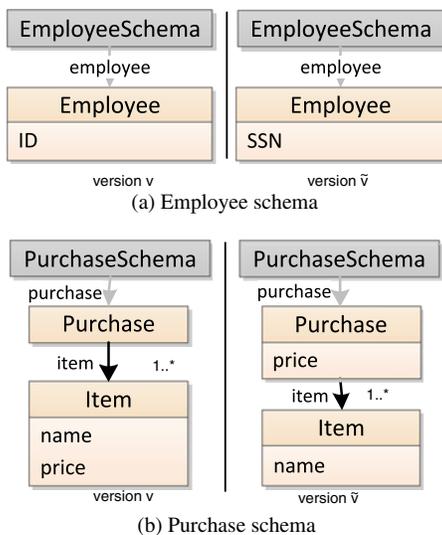


Fig. 7 Examples of schema evolution

These are examples of the problems that a non-trivial adaptation algorithm must solve. As we have mentioned in Section 5, when designing an adaptation algorithm, it is possible to take one of two distinct approaches: either (1) recording the changes as they are conducted during the schema evolution phase or (2) comparing two versions of the schema.

A system that records the changes usually provides some kind of a command that initiates the recording and after issuing this command all operations carried out by a user over the schema are recorded. When the recording is finished, the system can normalize the sequence, for example, by eliminating operations that cancel each other or by replacing groups of operations by other groups that lead to the same result but in a more straightforward way. These normalizing rules must be defined in the system. An alternative approach is to compare the two versions of the schema. The user can work with both schemas independently until (s)he is satisfied with them. The change detection algorithm then takes the schemas as an input and compares them. The result of the comparison is a list of differences between the schemas. Table 1 compares both the approaches from several aspects. In general, schema comparison has many advantages, but it must solve the problem of ambiguities of the sort as illustrated in the examples above.

If we do not want to settle for heuristics, the only correct solution is to find all possible interpretations and then let the user select the correct one. In our approach, we decided to solve this issue by adding another type of concepts into the model—*version links*. Version links connect constructs (i.e. classes, attributes etc.) that represent the same real-world concept in different versions of the model. They work as a mapping between different versions of the schema and allow us to distinguish whether a given concept in the new version is a completely new concept or whether it is an update of an existing one. Therefore, it enables us to avoid heuristics.

However, we do not rely upon the user—we do not need the user to specify the version links manually. In our framework, version links are most of the time kept and managed automatically in the background. Each time the user performs the branch operation, version links are created between all the concepts and their new versions. After that, they are maintained until a concept is deleted (then the version links from that concept must be removed as well). The user can add and remove version links manually (e.g. when (s)he is adding a new concept which should be mapped to old concept), but most of the time, they are managed by the system. One can regard version links as an

Table 1 Approaches to evolution: recording changes vs. schema comparison

Recording changes	Schema comparison
The recorded set should be normalized to eliminate redundancies (repeated changes in the same place etc.).	No need to look for redundancies; the set of changes is always minimal.
Once the evolution process is started, the old version cannot be easily changed.	Both old version and new version can be edited without limitations.
A user may want to interrupt his/her work at some point and continue in another session. The sequence of recorded changes would have to be stored and recording resumed later.	The process of evolution can be arbitrarily stopped and resumed.
When the user wants to retrieve the sequence for a reversed process, (s)he will have to either start with the new version and record the operations needed to go back to the old version again, or the system will have to be able to create an inverse sequence for each sequence of operations.	The reversed operation can be easily handled by the same algorithm, only with the two schemas on the input swapped.
When the evolved schema comes from an outer source, the sequence of operation changes cannot be retrieved directly; the user must start with his/her old version of the schema and manually adjust it to match the new schema.	A schema from an outer source can be imported into the system and serve as an input for the change detection algorithm.
The recorded set provides enough information to propagate changes in the schema to the documents; there are no ambiguities.	Without additional information, some type of changes cannot be distinguished (rename vs. add & remove, move vs. add & remove), methods for mapping discovery are necessary.

adoption from the change recording approaches and our approach can thus be considered as a combination of schema comparison (which is the core of the algorithm) and change recording (which maintains version links).

Definition 10 Let \mathcal{S}^* be a set of PIM schemas, \mathcal{S}' a set of PSM schemas. A *version links relation* is an equivalence relation $\mathcal{VL} \subset \mathcal{M}^* \times \mathcal{M}^*$ s.t. for any pair $(x, \tilde{x}) \in \mathcal{VL}$

- Both x and \tilde{x} are of the same kind (e.g. x is a PSM class (attribute,...) $\leftrightarrow \tilde{x}$ is a PSM class (attribute,...)) and
- $ver(x) \neq ver(\tilde{x})$.

We will call $(x, \tilde{x}) \in \mathcal{VL}$ *version link*.

A version link $(x, \tilde{x}) \in \mathcal{VL}$ specifies that both x and \tilde{x} represent the same schema or schema component but in different versions v and \tilde{v} , respectively. We will therefore say that x represents \tilde{x} in v or, symmetrically, \tilde{x} represents x in \tilde{v} . We will also simply say that x and \tilde{x} are *different versions of the same schema or schema component*.

We also introduce a partial function $getInVer$. Given a schema or schema component x and version v , the function returns a schema or schema component \tilde{x} which represents x in version v .

Definition 11 Function $getInVer : (\mathcal{M}^* \times \mathcal{V}) \rightarrow \mathcal{M}^*$ is defined as follows:

$$getInVer(x, v) = \tilde{x} \leftrightarrow$$

$$\exists \tilde{x} \in \mathcal{M}^* : (x, \tilde{x}) \in \mathcal{VL} \wedge ver(\tilde{x}) = v$$

For combinations of parameters (x, v) where no such \tilde{x} can be found, we will use the sign \perp in the meaning: $getInVer(x, v) = \perp \leftrightarrow \forall (x, \tilde{x}) \in \mathcal{VL} : ver(\tilde{x}) \neq v$.

Pairs for \mathcal{VL} are also added during operation *branch* (a new version \tilde{v} is created from version v), but can be changed by the user. Figure 8 shows the same schemas as Fig. 6, this time with version links as they were created when the schemas were branched and the second schema was edited (links between attributes were omitted for clarity, also, the figure does not show two version links between the schemas themselves). All the version links in the figure were created and maintained by the system, without the need from the user to interfere.

Example 1 When we go back to the pair of schemas depicted in Fig. 7a, with version links, we can unambiguously decide, which interpretation is correct. The second interpretation (attribute ID was renamed to SSN) is correct when there exists a version link (A', \tilde{A}') (where A', \tilde{A}' are the attributes named ID and SSN respectively). The usual process leading to this situation would be that the user created new version of

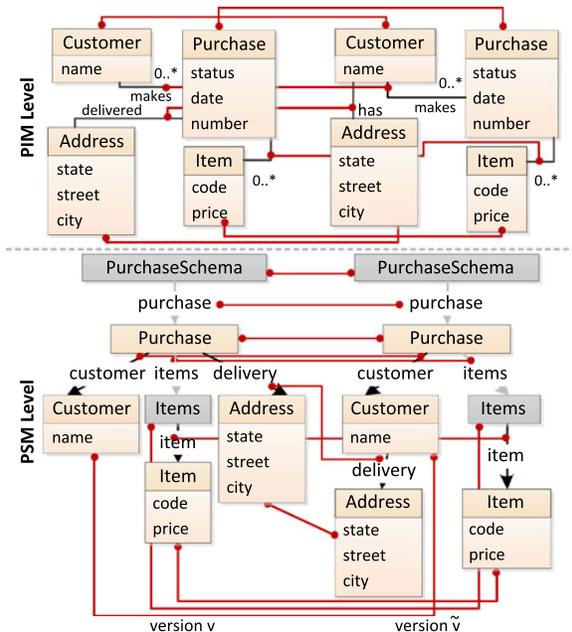


Fig. 8 Two versions of a PIM and a PSM schema with version links visualized

the schema via branch operation (which would create schema class $\tilde{E}S'$ as a copy of schema class ES' named *EmployeeSchema*, class \tilde{E}' as a copy of class E' named *Employee*, association $\tilde{A}E'$ as a copy of association AE' named *employee* and attribute \tilde{A}' as a copy of attribute A' named *ID*). Branch operation would create a copy of the schema and also version links $(ES', \tilde{E}S')$, $(AE', \tilde{A}E')$, (E', \tilde{E}') , (A', \tilde{A}') between the copied concepts. Then the user would rename attribute \tilde{A}' from *ID* to *SSN* in the second version (but link (A', \tilde{A}') to the attribute in the first version would be preserved). For the second example in Fig. 7b the situation is analogous and the interpretation would depend upon the existence of the version link between the attributes named *Item.price* and *Purchase.price* (moving an attribute also preserves the version link).

2 Changes

In Section 1.2, we introduced PIM and PSM schemas and the formalism for their versioning. We introduced the notion of version links which allow to trace how a given schema or schema component evolved in different versions. We can use the version links not only for tracing but also for adaptation of the underlying XML documents. It is possible to use version links at both PIM and PSM levels. We can also combine them with interpretations of the PSM schemas against the PIM schemas.

In this paper we, however, focus only on adaptation of XML documents which are instances of a single XML format modeled by a PSM schema. We suppose different version of the PSM schema and version links between the two versions. The rest of possibilities (i.e. exploiting also version links at the PIM level and interpretations of PSM schemas against the PIM schemas) is the matter of our future work.

In this section, we focus on possible kinds of changes between two versions of the PSM schema and its components. A change can be considered as a local difference between two PSM schemas (linked by a version link so they are different versions of each other). We distinguish a finite amount of types of changes (e.g. *classAdded*, *attributeMoved*, etc.). We then introduce the change detection algorithm which looks for particular changes of these types on the base of the version links.

As can be seen, we suppose that version links exist between two versions of a PSM schema. As we showed, this can be easily achieved when the user utilizes our framework which creates and maintains links as the user edits the versions. When a new version of the schema was not created using our framework (e.g. it was issued by a standardizing organization managing the specification which the modeled system adopted), the version links do not exist. To solve this problem, our framework supports reverse engineering and integration of schemas (Klímek and Nečaský 2010; Nečaský 2009). It maps new PSM schemas or their new versions to the PIM schema. The version links can be then deduced by composing the interpretations of the versions of the PSM schema against the PIM schema. However, various heuristics together with broader user interaction is required to create relation \mathcal{VL} . The possibility to infer version links from heuristics is not studied in this paper and is a part of our future work.

We can divide the set of types of changes which can occur between two versions of a PSM schema S' into four groups according to the character of a change (the classification is similar to Nečaský and Mlýnková (2009a)):

- Addition—a new construct was added to S' ,
- Removal—a construct was removed from S' ,
- Migratory—a construct (and possibly its subtree in S') was moved to another part of S' ,
- Sedentary—an existing construct in S' was adjusted in place, but not moved.

For each type there is also defined a type of construct where it can be detected. We call it *scope of change* or simply *scope*. There are four scopes of changes: *class*, *attribute*, *association*, and *content model*.

Table 2 Classification of changes

Change predicate	Category	Description
$classAdded(\tilde{C}', \tilde{R}')$	Addition	A new class \tilde{C}' is added as a child of association \tilde{R}' (if $\tilde{R}' = \perp$, \tilde{C}' is added as a new root class).
$classRemoved(C')$	Removal	Class C' is removed.
$classRenamed(\tilde{C}', \tilde{n}')$	Sedentary	The name of class \tilde{C}' is changed to $\tilde{n}' \in \mathcal{L}$. The name is mandatory for PSM classes, but can be changed.
$classMoved(\tilde{C}', \tilde{R}'_n)$	Migratory	Class \tilde{C}' is moved and becomes a child of association \tilde{R}'_n in version \tilde{v} (or becomes a new root class, in that case $\tilde{R}'_n = \perp$). This change encompasses changes of the <i>child</i> participant of associations (in contrast to <i>associationMoved</i> —see below).
$srIntroduced(\tilde{C}', \tilde{C}'_r)$	Sedentary	Class \tilde{C}' becomes a structural representative of another class \tilde{C}'_r in the schema. In the previous version, it was not a structural representative.
$srRemoved(\tilde{C}')$	Sedentary	Class \tilde{C}' is converted to a regular class. In the previous version, it was a structural representative.
$srChanged(\tilde{C}', \tilde{C}'_r)$	Sedentary	Class \tilde{C}' becomes a structural representative of another class \tilde{C}'_r in the schema. In the previous version, it was a structural representative of a different class.
$attributeAdded(\tilde{A}', \tilde{C}', \tilde{i}')$	Addition	A new attribute \tilde{A}' is added to class \tilde{C}' at position $\tilde{i}' \in \mathbb{N}_0$.
$attributeRemoved(A')$	Removal	Attribute A' is removed.
$attributeRenamed(\tilde{A}', \tilde{n}')$	Sedentary	The name of attribute \tilde{A}' is changed to $\tilde{n}' \in \mathcal{L}$.
$attributeMoved(\tilde{A}', \tilde{C}'_n, \tilde{i}')$	Migratory	The value of $class(\tilde{A}')$ is changed, i.e. attribute \tilde{A}' is moved from class \tilde{C}'_o to class \tilde{C}'_n at position $\tilde{i}' \in \mathbb{N}_0$. Moves within the same class are detected by <i>attributeIndexChanged</i> .
$attributeXFormChanged(\tilde{A}', \tilde{f}')$	Sedentary	The value of <i>xform</i> is changed from a to e or vice versa for attribute \tilde{A}' ($\tilde{f}' \in \{a, e\}$).
$attributeTypeChanged(\tilde{A}', \tilde{D}')$	Sedentary	The type of attribute \tilde{A}' is changed to $\tilde{D}' \in \mathcal{D}$.
$attributeIndexChanged(\tilde{A}', \tilde{i}')$	Migratory	Attribute \tilde{A}' is moved to position $\tilde{i}' \in \mathbb{N}_0$ within the same class as in version v . Moves between classes are detected by <i>attributeMoved</i> .
$attributeCardinalityChanged(\tilde{A}', \tilde{c}')$	Sedentary	The cardinality of attribute \tilde{A}' is changed to $\tilde{c}' \in \mathcal{C}$.
$associationAdded(\tilde{R}', \tilde{C}', \tilde{i}')$	Addition	A new association \tilde{R}' is added to the content of class \tilde{C}' at position $\tilde{i}' \in \mathbb{N}_0$.
$associationRemoved(R')$	Removal	Association R' is removed.
$associationRenamed(\tilde{R}', \tilde{n}')$	Sedentary	The name of association \tilde{R}' is changed to $\tilde{n}' \in \mathcal{L}$.
$associationMoved(\tilde{R}', \tilde{P}'_n, \tilde{i}')$	Migratory	Association \tilde{R}' is moved from the content of node \tilde{P}'_o to the content of node \tilde{P}'_n at position $\tilde{i}' \in \mathbb{N}_0$. This change encompasses changes of the <i>parent</i> participant of associations (in contrast to <i>classMoved</i> and <i>contentModelMoved</i> —see below).
$associationCardinalityChanged(\tilde{R}', \tilde{c}')$	Sedentary	The cardinality of association \tilde{R}' is changed to $\tilde{c}' \in \mathcal{C}$.
$associationIndexChanged(\tilde{R}', \tilde{i}')$	Migratory	Association \tilde{R}' is moved to position $\tilde{i}' \in \mathbb{N}_0$ (within the same class as in version v).
$contentModelAdded(\tilde{M}', \tilde{R}')$	Addition	A new content model \tilde{M}' is added as a child of association \tilde{R}' .
$contentModelRemoved(M')$	Removal	Content model M' is removed.
$contentModelMoved(\tilde{M}', \tilde{R}'_n)$	Migratory	Content model \tilde{M}' is moved and becomes a child of association \tilde{R}'_n in version \tilde{v} . Content models cannot be roots in a normalized PSM schema (see Definition 5). Thus, unlike <i>classMoved</i> , \tilde{R}'_n is never null for <i>contentModelMoved</i> .
$contentModelTypeChanged(\tilde{M}', \tilde{t}')$	Sedentary	The type of content model (<i>sequence, set, choice</i>) \tilde{M}' is changed to $\tilde{t}' \in \{\text{sequence, set, choice}\}$.

There are no predicates dedicated to the changes in the set S'_e and function *participant*, because each change in S'_e and *participant* is an inherent part of another change (*classAdded*, *classRemoved*, *classMoved*, *contentModelAdded*, *contentModelRemoved*, *contentModelMoved*, *associationAdded*, *associationRemoved*). Thus, changes in S'_e and *participant* are detected and documents adapted within the scope of the changes listed above

2.1 Change predicates

A change predicate is a formalization of a certain type of change between two versions of a PSM schema. Each change predicate has a certain amount of parameters. Change detection can be then formalized as looking for n -tuples satisfying the change predicates. The first parameter always corresponds to one of the scopes. We will use c_{ins} to denote the set of all n -tuples which satisfy a change predicate c . We will call it the *set of instances of c* .

Table 2 contains all the change predicates grouped by the scope with their respective categories and description. We suppose a PSM schema in two different versions $v, \tilde{v} \in \mathcal{V}$. We will use tilde to mark constructs that belong to $\mathcal{M}^*[\tilde{v}]$, constructs without the tilde mark belong to $\mathcal{M}^*[v]$. I.e. $S' = (S'_c, S'_a, S'_r, S'_m, C_{S'}) \in \mathcal{M}^*[v]$ denotes the PSM schema in version v and $\tilde{S}' = (\tilde{S}'_c, \tilde{S}'_a, \tilde{S}'_r, \tilde{S}'_m, \tilde{C}'_{S'}) \in \mathcal{M}^*[\tilde{v}]$ denotes the PSM schema in version \tilde{v} .

Example 2 For an example of change predicates, let us go back to Fig. 7. Let us assume a version link between attributes A' named ID and \tilde{A}' named SSN in Fig. 7a. Table 2 contains a change predicate *attributeRenamed* with parameters $\tilde{A}' \in \tilde{S}'_a$ and $\tilde{n}' \in \mathcal{L}$ (the new name). Statement $(\tilde{A}', "SSN") \in \text{attributeRenamed}_{ins}$ is a formal expression of the fact that the name of the attribute was changed from ID to SSN.

Similarly, let us assume a version link between A'_p and \tilde{A}'_p (i.e. attributes `Item.price` and `Purchase.price` in Fig. 7b, where $ver(A'_p) = v$ and $ver(\tilde{A}'_p) = \tilde{v}$). Let \tilde{C}'_p be the class named `Purchase` in version \tilde{v} . Table 2 contains a change predicate *attributeMoved* with parameters $\tilde{A}' \in \tilde{S}'_a, \tilde{C}'_n \in \tilde{S}'_c$ and $\tilde{i}' \in \mathbb{N}_0$. Statement $(\tilde{A}'_p, \tilde{C}'_p, 0) \in \text{attributeMoved}_{ins}$ is a formal expression of the fact that attribute `price` was moved to class `Purchase` to the position 0.

For the purposes of implementation of the change detection and adaptation algorithms, we defined each change predicate formally; however, due to space limitations, we will not include the formal definitions of all the change predicates in this paper. We selected three change predicates—*attributeAdded*, *associationIndexChanged* and *classMoved*—for demonstration:

$$\begin{aligned} & \tilde{A}' \in \tilde{S}'_a \wedge \tilde{C}' \in \tilde{S}'_c \setminus \{\tilde{C}'_{S'}\} \wedge \tilde{i}' \in \mathbb{N}_0 \wedge \\ & \text{getInVer}(\tilde{A}', v) = \perp \\ & \wedge \text{position}(\tilde{A}', \text{attributes}(\tilde{C}')) = \tilde{i}' \\ & \Leftrightarrow \text{attributeAdded}(\tilde{A}', \tilde{C}', \tilde{i}') \end{aligned} \tag{8}$$

$$\begin{aligned} & (\tilde{R}' \in \tilde{S}'_r \wedge \tilde{i}' \in \mathbb{N}_0 \wedge C'_1 \in \mathcal{N}' \wedge \tilde{C}'_1 \in \tilde{\mathcal{N}}' \wedge \\ & \text{getInVer}(\tilde{R}', v) \neq \perp \wedge C'_1 = \text{getInVer}(\tilde{C}'_1, v) \wedge \\ & \text{parent}(\text{getInVer}(\tilde{R}', v)) = C'_1 = \\ & \text{getInVer}(\text{parent}(\tilde{R}'), v) \wedge \\ & \tilde{i}' \neq \text{position}(R', \text{content}(C'_1)) \wedge \\ & \tilde{i}' = \text{position}(\tilde{R}', \text{content}(\tilde{C}'_1)) \\ & \Leftrightarrow \text{associationIndexChanged}(\tilde{R}', \tilde{i}') \end{aligned} \tag{9}$$

$$\begin{aligned} & \tilde{C}' \in \tilde{S}'_c \setminus \{\tilde{C}'_{S'}\} \wedge \tilde{R}'_n \in \tilde{S}'_r \wedge \\ & \text{getInVer}(\tilde{C}', v) = C' \neq \perp \wedge \text{child}(\tilde{R}'_n) = \tilde{C}' \wedge \\ & [(\exists P'_o A \in S'_r)(\text{child}(R'_o) = C') \wedge \\ & P'_o A \neq \text{getInVer}(\tilde{R}'_n, v) \\ & \vee (\forall P'_o A \in S'_r)(\text{child}(R'_o) \neq C')] \\ & \Leftrightarrow \text{classMoved}(\tilde{C}', \tilde{R}'_n) \end{aligned} \tag{10}$$

$$\begin{aligned} & \tilde{C}' \in \tilde{S}'_c \setminus \{\tilde{C}'_{S'}\} \wedge \text{getInVer}(\tilde{C}', v) = C' \neq \perp \\ & \wedge (\exists R'_o A \in S'_r)(\text{child}(R'_o) = C') \wedge \\ & (\forall \tilde{R}'_n \in \tilde{S}'_r)(\text{child}(\tilde{R}'_n) \neq \tilde{C}') \\ & \Leftrightarrow \text{classMoved}(\tilde{C}', \perp) \end{aligned} \tag{11}$$

Predicate 8 says that the examined attribute \tilde{A}' has no counterpart A' present in version v and the position of \tilde{A}' among the attributes of class \tilde{C}' equals to \tilde{i}' . Predicate 9 says that the parent of the examined association \tilde{R}' has not changed between versions v and \tilde{v} , but the position of \tilde{R}' in the content of its parent has changed to \tilde{i}' . Predicate 10 says that the examined class \tilde{C}' was moved under association \tilde{R}'_n either from the root or from another association, whereas Predicate 11 says that \tilde{C}' was moved from an association $P'_o A$ to a root.

With the formal definitions of change predicates, we are able to detect differences between two compared versions of a schema. Now we can describe how algorithm *DetectChanges* (see Algorithm 1 for pseudo-code listings). It takes as an input two versions of the PSM schema: S' and S' (s.t. $ver(S') = v$ and $ver(S') = \tilde{v}$) and the relation $\mathcal{V}\mathcal{L}$. For each change predicate, it examines all constructs in the appropriate scope and tests, whether the predicate is satisfied for any combination of other parameters. Although the description of the algorithm may arise a suspicion of inefficiency, it is possible to define a more efficient lookup subroutine for each change predicate. For instance, for predicate *classMoved*, it is not necessary to test all associations

for parameter \tilde{R}'_n , but only the actual parent of class \tilde{C}' . That is how the actual implementation works.

The output of the algorithm is the set $C_{S',\tilde{S}',v,\tilde{v}}$ $\cup (c, c_{\text{ins}})$, containing for each change predicate the set of all of its instances. The output set $C_{S',\tilde{S}',v,\tilde{v}}$ captures the changes made between the two versions of a PSM schema.

Algorithm 1 DetectChanges

Input: old and new version $v, \tilde{v} \in \mathcal{V}$, PSM schemas S', S'

Output: $C_{S',\tilde{S}',v,\tilde{v}}$ —set of changes between S' and S'

```

1:  $C_{S',\tilde{S}',v,\tilde{v}} \leftarrow \emptyset$ 
2: for all change predicate  $c$  do
3:    $c_{\text{ins}} \leftarrow \emptyset$ 
4: end for
5: for all change predicate  $c$  of arity  $k$  do
6:   for all tuple  $t \in (S'_{\text{all}} \times \tilde{S}'_{\text{all}})^k$  do
7:     if  $c(t)$  then {tuple  $t$  satisfies  $c$ }
8:        $c_{\text{ins}} \leftarrow c_{\text{ins}} \cup t$ 
9:     end if
10:  end for
11:   $C_{S',\tilde{S}',v,\tilde{v}} \leftarrow C_{S',\tilde{S}',v,\tilde{v}} \cup \{(c, c_{\text{ins}})\}$ 
12: end for

```

2.2 Impact on validity

The output set $C_{S',\tilde{S}',v,\tilde{v}}$ of algorithm *DetectChanges* can be further analyzed. Having detected the set of change instances $C_{S',\tilde{S}',v,\tilde{v}}$, we can determine the impact of evolution on validity. Some change instances do not affect validity of the documents in $\mathcal{T}(S')$. However, they may affect other parts of our five-level framework. For example, during the translation of the PSM schema into XSD, class names are used for naming the generated complex types. Renaming a class does not invalidate $\mathcal{T}(S')$ (because class names do not correspond to content of the documents), but the XSD generated from S' will differ from the XSD translated from S' .

The ability to identify instances of change predicates not affecting validity can significantly simplify the process of XML document adaptation. Of course, if we are sure that all the detected change instances in the evolved schema do not affect validity, it is correct to skip the adaptation of $\mathcal{T}(S')$, because validity against the new schema is guaranteed. For each change predicate, we can define additional tests that, if satisfied, ensure that the instance of the change predicate does not affect validity of the documents from $\mathcal{T}(S')$.

Definition 12 Let c be a change predicate (as listed in Table 2) and $i_c \in c_{\text{ins}}$ its instance. We define predicate c^{NI} , called *NI-predicate* for c , with the same parameters as change predicate c . When c^{NI} is satisfied for an instance i_c , this instance does not affect validity of documents in $\mathcal{T}(S')$.

Example 3 shows several NI-predicates and Lemma 1 joins NI-predicates with the notion of backwards compatibility from Definition 9. Its proof is a direct application of the definitions.

Example 3 As an example consider the following NI-predicates:

$$\text{classRenamed}^{NI}(\tilde{C}', \tilde{n}') \leftrightarrow \text{true} \tag{12}$$

$$\begin{aligned} \text{attributeCardinalityChanged}^{NI}(\tilde{A}', \tilde{m}'.. \tilde{n}') \leftrightarrow \\ \text{getInVer}(\tilde{A}', v) = A' \wedge \text{card}(A') = m'..n' \wedge \\ m' \geq \tilde{m}' \wedge n' \leq \tilde{n}' \end{aligned} \tag{13}$$

$$\begin{aligned} \text{associationIndexChanged}(\tilde{R}', \tilde{i}') \leftrightarrow \\ \text{getInVer}(\tilde{R}', v) = R' \wedge \\ \text{parent}(\tilde{R}') = \tilde{M}' \in \tilde{S}'_m \wedge \text{parent}(R') = M' \in S'_m \wedge \\ ((\text{cmtype}(M') \in \{\text{sequence}, \text{set}\} \wedge \\ \text{cmtype}(\tilde{M}') = \text{set}) \vee \\ (\text{cmtype}(M') = \text{choice} \wedge \\ \text{cmtype}(\tilde{M}') = \text{choice})) \end{aligned} \tag{14}$$

Predicate 12 is satisfied for all instances of classRenamed, because the name of a class does not correspond to any part of the modeled XML document. All instances of *classRenamed* thus do not violate validity. Predicate 13 is satisfied for those instances of *attributeCardinalityChanged* which broaden the cardinality interval. Predicate 14 is satisfied for those instances of *associationIndexChanged* which reorder content of content models of type *set* and *choice*. For these, the ordering of content is not significant.

Predicates for other changes are defined in a similar manner (and, of course, predicates for some changes are never satisfied, because the change always violates validity).

Lemma 1 Let S' and S' be two versions of a PSM schema ($\text{ver}(S') = v, \text{ver}(S') = \tilde{v}$). Let $C_{S',\tilde{S}',v,\tilde{v}}$ be

the output set of algorithm DetectChanges for these schemas. Then:

$$(\forall(c, c_{\text{ins}}) \in C_{S', \tilde{S}', v, \tilde{v}})(\forall i \in c_{\text{ins}})(c^{NI}(i_1, \dots, i_k) \rightarrow \mathcal{T}(S') \subseteq \mathcal{T}(\tilde{S}'))$$

where i_1, \dots, i_k are elements of k -tuple $i \in c_{\text{ins}}$ with arity k .

In other words, if c^{NI} is true for all the instances of every predicate c , then S' is backwards compatible.

3 Adaptation

When the new version S' of the schema S' invalidates the set $\mathcal{T}(S')$, we need to adapt the documents respectively. For each change predicate, we describe how documents in $\mathcal{T}(S')$ should be adapted. We will describe adaptation as a function *adapt* with the following semantics:

$$\forall T \in \mathcal{T}(S') \setminus \mathcal{T}(\tilde{S}') : \text{adapt}(T) \in \mathcal{T}(\tilde{S}') \quad (15)$$

$$\forall c, \forall i, i \in c_{\text{ins}}, (c, c_{\text{ins}}) \in C_{S', \tilde{S}', v, \tilde{v}} : \text{instance } i \text{ is adapted correctly in } \text{adapt}(T) \quad (16)$$

The first condition defines correctness w.r.t. to the evolved schema, i.e. that the adapted document is valid against the new version. The second condition defines correctness w.r.t. the detected set of changes. It must be pointed out that not every action, that formally makes a document valid, can be considered a correct adaptation. For instance, let a user move an optional attribute in a PSM schema from its current class to another class. Deleting the corresponding parts in the document would not be the correct adaptation even though the result is formally valid. We need a more sophisticated adaptation which correctly moves the corresponding parts in the document.

Correct adaptations for each change predicate are described in the rest of this section. The *adapt* function behaves differently for each change predicate and has different preconditions. For some predicates the function has more alternative behaviors depending on various conditions which we discuss in the following text. However, any document which is valid against the old version of the schema can be adapted and the adaptation results into a document which is valid against the new version of the schema.

We gave reasons for using a comparison-based approach in Table 1 and the paragraphs below. Our approach can be compared to a concrete incremental approach. If we are able to identify for each evolution

primitive of the incremental approach (e.g. *remove element with simple content*) a change predicate with the same semantics (e.g. *attributeRemoved*), we can simulate the incremental approach with ours (by performing the whole adaptation cycle after each change). The strength of the incremental approaches thus lies in the number of supported evolution primitives. Because this number is usually small (see Section 5) and our approach uses some change predicates, for which there is no corresponding evolution primitive, our approach is stronger.

We do not expect any specific implementation language (Kay 2007; W3C 2011, 2004; ISO 2008) in our description. The sketch of the implementation (the creation of an adaptation script) using XSLT can be found in Malý et al. (2011).

3.1 Class changes

classAdded (\tilde{C}' , \tilde{R}') If the added class \tilde{C}' is a *top* class (i.e. child of schema class C'_S), Definition 5 requires the association between C'_S and \tilde{C}' to have a name. If its whole subtree $\Delta\tilde{C}'$ was added in \tilde{v} , i.e. ($\forall \tilde{x} \in \Delta\tilde{C}'(\text{getInVer}(\tilde{x}, v) = \perp)$), we do not have to adapt anything, because the added subtree defines whole new possible shape of a valid document. Similarly, when \tilde{C}' is not a top class, but $\text{card}(\tilde{R}') = 0..n$, and again $\Delta\tilde{C}'$ was added in \tilde{v} , the change does not require adaptation (the added subtree defines a whole new *optional* part of a document).

In other cases, an instance of the class \tilde{C}' is created during adaptation. These are the cases where the user decided to add a new class to refine the structure of the document. (S)he may have decided to, e.g., add a wrapping element for some elements to increase readability (in that case, most likely the added class will not have an interpretation against the PIM schema). If $\text{name}(\tilde{R}') \neq \lambda$, creating an instance of \tilde{C}' means adding an XML element into the document. If the user adds a new, non-optional class with semantic meaning (interpretation $I(\tilde{C}')$ is defined), the adaptation remains the same, an element is created if $\text{name}(\tilde{R}') \neq \lambda$.

If \tilde{C}' or \tilde{C}' itself are referenced by a structural representative (this is guaranteed by condition 5 from Definition 5 of a normalized PSM schema), each such reference will be processed separately, following the same principles as described above.

classRemoved (C') Removal of a construct from the model must be always solved by removal of the content modeled by the removed construct from the documents in $\mathcal{T}(S')$. However, the content modeled by the whole subtree $\Delta\tilde{C}'$ cannot be instantly removed from the

document, because some other changes may move parts of this content to other parts of the document.

classRenamed (\tilde{C}', \tilde{n}') This change does not require any adaptation of XML documents, because the name of a PSM class is not reflected in the XML document.

Translation of PSM schemas to XML Schema uses class names to name complex types, groups and attribute groups, so changing the name of a class results in changing the name of a complex type in the XSD. If names of types, groups and attribute groups in XSD need to remain consistent with names of constructs in other components of the systems (i.e. with names of tables and columns in a relational database or names of classes in an object model), these construct should be renamed too.

3.2 Attribute changes

attributeAdded ($\tilde{A}', \tilde{C}', \tilde{i}'$) If attribute \tilde{A}' is added as mandatory, a new content must be added into the document—either an XML element with a simple content or an XML attribute (if $xform(\tilde{A}') = e$ or a respectively). More about generating content can be found in Section 3.7.

attributeRemoved (A') All instances of attribute A' (i.e. XML attributes or XML elements with simple content) must be removed from the document.

attributeRenamed (\tilde{A}', \tilde{n}') Each XML attribute/element modeled by $A' = getInVer(\tilde{A}', v)$ (named $name(A')$) must be renamed to \tilde{n}' in the XML document.

attributeXFormChanged (\tilde{A}', \tilde{f}') Changing the *xform* of a PSM attribute A' requires:

- Creating a new XML node of the respective type in the new location—either a new XML attribute, if $xform(\tilde{A}') = a$, or an XML element, if $xform(\tilde{A}') = e$. The node value is copied from the old instance.
- Deleting the instance of A' from its previous location (It can be either an XML attribute, if $xform(A') = a$, or an XML element with simple content, in that case $xform(A') = e$.)

attributeTypeChanged (\tilde{A}', \tilde{D}') Let D' be the type in version v , i.e. $D' = type(getInVer(\tilde{A}', v))$. Adaptation of documents may be skipped in case when $D' \subseteq \tilde{D}'$. This condition is guaranteed if D' is a type derived from \tilde{D}' using restriction. The condition means that the requirements for the documents were relaxed and a more general set of values is allowed. In the opposite situation, instead of a general set of values, the require-

ments are made more strict and only a specific subset of values is allowed. E.g. instead of an arbitrary string for `email` attribute in the old version, only strings valid against a regular expression describing all the possible email addresses are allowed in the new version. In such case $D' \supseteq \tilde{D}'$. In the general case the two sets are incomparable.

Let us denote $[A'][\mathcal{T}(S')]$ the set of all values of attribute A' in all documents in $\mathcal{T}(S')$. Then we can extend the previous approach if $[A'][\mathcal{T}(S')] \subseteq \tilde{D}'$. In this case no adaptation is needed again. Verifying this condition cannot be possible in every case; however, in some situations, it can be done easily. For instance, when we return to the email example, the XML schema may define an email as an arbitrary string, but the system contains another component that verifies each email more strictly, before it can occur in a document $D \in \mathcal{T}(S')$. The applicability of this approach can be decided by the user.

If adaptation is really necessary, function $conv_{A'} : D' \rightarrow \tilde{D}'$ or (since we do not need to be able to convert all the possible values in D') $conv_{A'} : [A'][\mathcal{T}(S')] \rightarrow \tilde{D}'$ must be provided for the adaptation algorithm.

Function $conv_{A'}$ can be reused by pairs of attributes with the same pairs of types, i.e. for attributes $(A', \tilde{A}') \in S'_a \times \tilde{S}'_a$ s.t. $type(A') = D'$ and $type(\tilde{A}') = \tilde{D}'$, function $conv_{A'} = conv_{D', \tilde{D}'}$ converting values from the domain of D' to values from the domain of \tilde{D}' can be used.

Alternatively, the function can be defined separately for each attribute A' .

attributeIndexChange (\tilde{A}', \tilde{i}') Adaptation depends on the values of $f' = xform(A')$ and $\tilde{f}' = xform(\tilde{A}')$. If either $f' = a$ or $\tilde{f}' = a$, the attribute modeled an XML attribute in the old version or does so in the new version. Since the order of attributes in an XML element is insignificant and applications should not rely on the order of attributes, no adaptation is needed.

If both $f' = \tilde{f}' = e$ the order of attributes determines the order of XML subelements, which is significant. The change then requires reordering of the subelements modeled by the attributes with respect to the new order of *attributes* (\tilde{C}').

attributeCardinalityChange (\tilde{A}', \tilde{c}') Let $card(A') = (\tilde{m}', \tilde{n}')$, $getInVer(\tilde{A}', v) = A'$ and $card(A') = m'..n'$. For cardinality changes, there are two adaptation actions from which none, one or both must be undertaken to adapt a document (varying from document to document).

- If $\tilde{m}' > m'$, new content may have to be added for some documents.

- If $\tilde{n}' < n'$, content may have to be removed from some documents.

For each document $D \in \mathcal{T}(S')$, the number of XML nodes (elements or attributes, depending on the value of $xform(A')$) that are instances of A' differs (unless $m' = n'$), therefore the amount of XML nodes that need to be added/removed differs too.

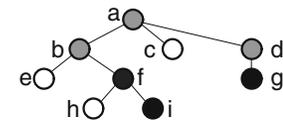
When removing nodes, the algorithm must either choose which nodes to keep and which to delete (one solution can be to always keep those nodes that occur earlier in the document) or leave this choice up to the user.

When adding nodes, the values for these nodes must be assigned. Raising the lower cardinality from $m' \geq 1$ to $\tilde{m}' > m'$ raises the minimum allowed occurrences, (the special case $m' = 0$ and $\tilde{m}' \geq 1$ makes an optional subelement/attribute mandatory). That is why approaches to generate values of attributes need to be discussed. (In this particular case, a suitable solution would be using a default value of the attribute—see Section 3.7).

attributeMoved ($\tilde{A}', \tilde{C}'_n, \tilde{v}$) Moving an attribute is an evolution operation that requires more in-depth enquiry. The aim of our approach is to keep the semantics of the adapted document and not to lose the existing data during adaptation. The trivial solution—deleting the attribute from its former location in the document and creating a new attribute in the new location (as used in Su et al. (2002) and Guerrini et al. (2007))—is not suitable, because the value of the attribute is lost. The most general approach is to couple each instance ($\tilde{A}', \tilde{C}'_n, \tilde{v}$) of *attributeMoved* change with an adaptation function $attMove_{\tilde{\chi}}(oldLocations, newLocation)$ where *oldLocations* selects all existing instances of A' and *newLocation* contains the new location of the instance in the adapted document. The result of the function is the new value for the instance of \tilde{A}' in the new document. In general, the function $attMove_{\tilde{\chi}}$ is defined by the user, but the system can provide the user with a suggestion in certain cases—several types of the most common scenarios can be distinguished.

In the following text we expect that the attribute was moved between classes C'_o and \tilde{C}'_n , i.e. attribute $A' \in attributes(C'_o)$, $\tilde{A}' \in attributes(\tilde{C}'_n)$. Let $\tilde{C}'_o = getInVer(C'_o, \tilde{v})$ and $C'_n = getInVer(\tilde{C}'_n, v)$ be the new version of class C'_o and the old version of class \tilde{C}'_n respectively, both can be \perp . In the situation depicted in Fig. 7b $A' = price_1$, $\tilde{A}' = price_2$, $C'_o = Item_1$, $C'_n = Purchase_1$, $\tilde{C}'_o = Item_2$, $\tilde{C}'_n = Purchase_2$ (we use subscripts to distinguish constructs in version v and \tilde{v}). We will use an auxiliary function *tree*, that returns the smallest tree that contains a set of nodes. Formally:

Fig. 9 Example of *tree* function



Definition 13 For a rooted tree (V, E) , $tree(X)$, $tree : 2^V \rightarrow 2^{(V \cup E)}$ returns the nodes and edges of the subgraph of the smallest common subtree for a set $X \subseteq V$, containing root b of the common subtree, members of X and for each $n \in X$ path between n and the root b .

Example 4 An example of the result of function *tree* is depicted in Fig. 9. The result of $tree(\{f, g, i\})$ is the set $\{a, b, d, f, g, i, a - b, b - f, f - i, a - d, d - g\}$, where a is the root of the common subtree and $a - b$ the edge from a to b , etc.

In addition, we define predicate $stable(\mathcal{X}')$, $\mathcal{X}' \subseteq \{S'_{all} \setminus S'_a\} \leftrightarrow \forall X' \in \mathcal{X}' : getInVer(X', \tilde{v}) = \tilde{X}' \neq \perp$ and \tilde{X}' was not moved, added or deleted and its cardinality was not changed (if X' is an association).

The intuitive meaning of predicate $stable(\mathcal{X}')$ is that there were no radical changes in the structure of the schema regarding for the members of \mathcal{X}' . If $C'_n \neq \perp$, $T' = tree(\{C'_o, C'_n\})$ and $stable(T')$ holds, then:

- If $\forall association R' \in T' : card(R') = m_{R'}.1$ (i.e. only cardinalities 0..1 and 1..1 are allowed in the affected part of the schema) and, therefore, the attribute \tilde{A}' will have 0 or 1 instance in the old schema, then this instance can be copied to the only one new location ($attMove_{\tilde{\chi}}$ will be *identity* function).
- If C'_o is a descendant of C'_n in the PSM tree (the attribute is moved upwards, but the associations between C'_o and C'_n can have arbitrary cardinalities), then all instances of A' under each instance of C'_n should be “aggregated” to one instance of \tilde{A}' . Several aggregation functions can be offered (e.g. *sum*, *count*, *avg*, *max*, *min* known from relational databases or *concat* inlining the respective values).
- If C'_o is a descendant of \tilde{C}'_n in the PSM tree, $card(A') = m'.1$ and $card(\tilde{A}') = \tilde{m}'.*$, then this case is similar as the case above, but the cardinality of attribute \tilde{A}' is adjusted, so all the values from existing instances can be used as values of \tilde{A}' . No aggregation is needed.
- If C'_o is an ancestor of C'_n in the PSM tree, $card(A') = m'.*$ and $card(\tilde{A}') = m'.1$, then this is an inverse case to the one above. The respective values of A' can be distributed to the new locations. Nonethe-

less, a user may have to specify the distribution precisely.

When none of the conditions above is satisfied, a possible general approach is to use the function $attMove_{\tilde{A}'} = identityN$ which returns the value of the n -th instance of A' when required at the n -th location in the adapted document. Other $attMove_{\tilde{A}'}$ functions must be provided by the user.

3.3 Association changes

In the following text, let $\tilde{R}' = (\tilde{E}'_1, \tilde{E}'_2) \in \tilde{S}'_r$ be a PSM association, $R' = (E'_1, E'_2)$ its previous version (if it exists), $participant(\tilde{E}'_1) = \tilde{C}'_1$, $participant(\tilde{E}'_2) = \tilde{C}'_2$, $participant(E'_1) = C'_1$, $participant(E'_2) = C'_2$.

associationAdded ($\tilde{R}', \tilde{C}', \tilde{i}$) If $name(\tilde{R}')$ is defined (association has a name $\tilde{n}' \in \mathcal{L}$), wrapper XML element named \tilde{n}' will be put to the adapted document and then the adaptation proceeds to adapt the child node $\tilde{C}'_2 = child(\tilde{R}')$. If \tilde{C}'_2 is a construct added in the new version ($getInVer(\tilde{C}'_2, v) = \perp$), adaptation is performed within the scope of adaptation of *classAdded/contentModelAdded* change described later in this section. Otherwise (when $getInVer(\tilde{C}'_2) = C'_2 \neq \perp$), C'_2 was moved from its previous location in the PSM schema tree. In that case, adaptation is performed within the scope of adaptation of *classMoved/contentModelMoved* change.

associationRemoved (R') If $name(R')$ is defined, the matching wrapping XML element is removed. Depending on whether $child(R') = C'_2$ was deleted or not (i.e. it was moved), the adaptation continues within the scope of adaptation of *classRemoved/contentModelRemoved* or *classMoved/contentModelMoved* changes, respectively.

associationCardinalityChange (\tilde{R}', \tilde{c}') Similarly as with *attributeCardinalityChanged*, there exist two adaptation actions, from which none, one or both must be undertaken to adapt a document (varying from document to document).

Let $card(R') = m'..n'$ and $card(getInVer(\tilde{R}', v)) = card(\tilde{R}') = (\tilde{m}', \tilde{n}')$.

- If $\tilde{m}' > m'$, new content may have to be added for some documents.
- If $\tilde{n}' < n'$, content may have to be removed from some documents.

In case of PSM attributes, the content added or deleted involves either XML attribute or leaf XML elements with simple content. With PSM association the adaptation actions have to deal with whole XML subtrees.

For each document $D \in \mathcal{T}(S')$ the number of XML nodes (elements or attributes, depending on the value of $xform(R')$) that are instances of R' differs (unless $m' = n'$). Therefore the amount of XML nodes that need to be added/removed differs too.

When removing nodes, the algorithm must either choose which nodes to keep and which to delete (one solution can be always keep those nodes that occur earlier in the document) or leave this choice up to the user.

When adding, the content for the new instances must be generated (this involves generating a whole XML subtree). More about generating content can be found in Section 3.7.

associationIndexChange (\tilde{R}', \tilde{i}') Two different cases can be distinguished for *associationIndexChanged*: either (1) $\{C'_1, \tilde{C}'_1\} \subseteq S'_c \cup \{M' : S'_m | cmtree(M') = sequence\}$ or (2) at least one of C'_1 and \tilde{C}'_1 is a content model and $cmtree(\tilde{C}'_1) \in \{choice, set\}$. In the former case, adaptation is needed and content modeled by \tilde{C}'_2 must be moved to the proper location. There is only one exception. When *subtree* \tilde{C}'_2 models only XML attributes and no XML elements, no adaptation is needed, because the order of attributes is not significant in the XML data model and no application should rely on attributes being defined in some particular order. In the latter case, no adaptation is necessary, because the ordering of content is irrelevant in choices/sets.

associationRenamed (\tilde{R}', \tilde{n}') Since λ values must be taken into consideration, three model cases can be distinguished. Let $n' = name(getInVer(\tilde{R}'))$:

- If $n' = \lambda \wedge \tilde{n}' \neq \lambda$, the association was given a name, which means the each instance (since R' of R' can have cardinality $\neq 1..1$) will be wrapped in a new XML element with name \tilde{n}' . If subtree $\Delta C'_2$ models some attributes with $xform = a$, their instances will now be moved to attributes of the wrapping XML element.
- If $n' \neq \lambda \wedge \tilde{n}' \neq \lambda$, the association is renamed, which means each wrapping XML element modeled by R' will be renamed to \tilde{n}' .
- If $n' \neq \lambda \wedge \tilde{n}' = \lambda$, the name is removed from an association, the needed adaptation is an exact opposite of the first case, which means that the wrapping XML element is removed (and when it contains some XML attributes they are moved upwards).

3.4 Content model changes

contentModelAdded, *contentModelRemoved* Adaptation of these two changes follows the same principles as adaptation of classAdded and classRemoved changes.

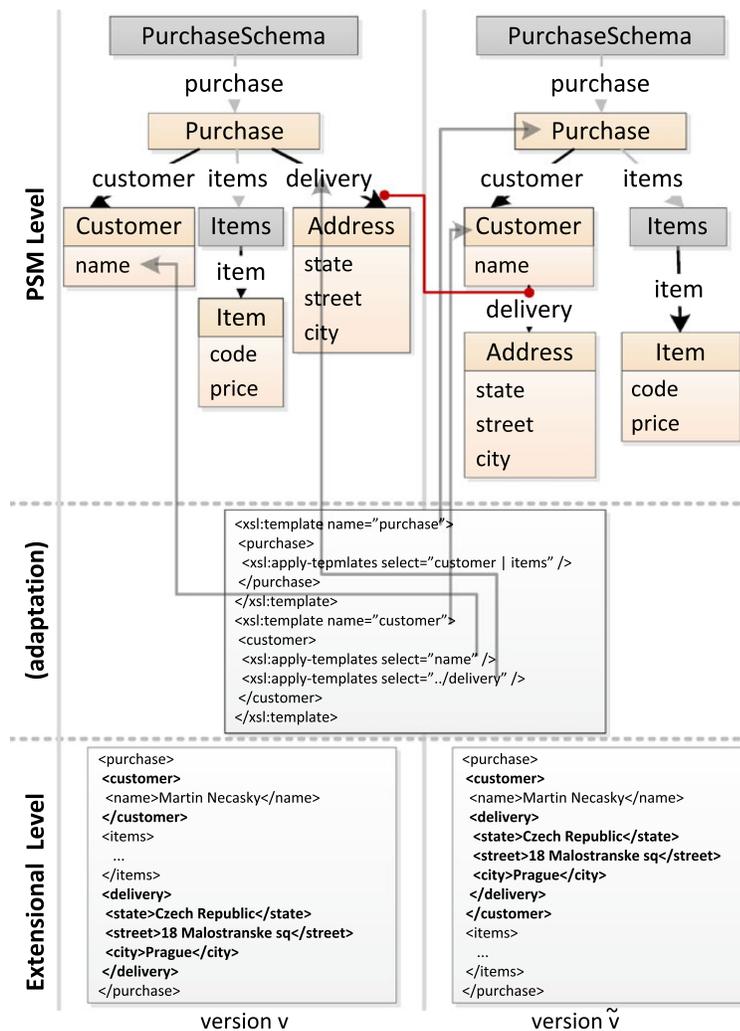
contentModelTypeChanged (\tilde{M}', \tilde{t}') In this part, let $L' = childNodesOf(M')$, $\tilde{L}' = childNodesOf(\tilde{M}')$. The list \tilde{L}' may contain three groups of nodes:

1. Nodes added in version \tilde{v} ,
2. Nodes whose counterparts in version v are members of the list L' , and
3. The rest—nodes whose counterparts in version v reside elsewhere in the PSM tree.

Nodes from groups 2 and 3 may have instances in the document D . On the basis of values $\tilde{t}' = cmttype(\tilde{M}')$ and $t' = cmttype(M')$ we can distinguish the following situations:

- If $t' \in \{sequence, set\} \wedge \tilde{t}' = choice$, when processing an instance of M' , one child node \tilde{C} from \tilde{L}' must be selected and instance of \tilde{C} will be included in the adapted document. Groups 2 and 3 are preferred when selecting the node C . If there are more candidates, it is up to the user to make the decision.
- If $t' = sequence \wedge \tilde{t}' = set$, no adaptation needed, because set is more relaxed than sequence.
- If $t' = choice \wedge \tilde{t}' \in \{sequence, set\}$, a content must be added for each member of \tilde{L}' which is

Fig. 10 associationMoved—Revalidation



not optional and no instance was found for it in document D .

- If $t' = \text{set} \wedge \tilde{t}' = \text{sequence}$, instances must be re-ordered to follow the ordering of \tilde{L}' .

3.5 Changes moving classes, content models and associations

associationMoved ($\tilde{R}', \tilde{P}'_n, \tilde{t}'$) The content modeled by R' will be removed from the processed instance of P'_o . Since in the new version, \tilde{R}' is among contents of \tilde{P}'_n , the wrapping XML subelement is created (if \tilde{R}' has a name) in the instance of \tilde{P}'_n .

If $\text{getInVer}(\text{child}(\tilde{R}'), v) = \text{child}(R')$, i.e. the association was moved with its child (which is the usual situation, but not the general case), the adaptation proceeds to the child, i.e. the instances of $\text{child}(R')$ will be converted to instances of $\text{child}(\tilde{R}')$. The situation is similar to adaptation of *attributeMoved* change described in Section 3.2. The adaptation is again largely affected by the cardinalities of the concerned association and the positions of the nodes P'_o and \tilde{P}'_n . In case of $\text{getInVer}(\tilde{P}'_n, v) = P'_n \neq \perp$ and $\text{stable}(\text{tree}(P'_o, P'_n))$, we can distinguish some cases corresponding to those proposed for *attributeMoved* and offer similar options for adaptation. Otherwise, the user must provide his/her adaptation function for the particular case.

Nonetheless, one additional aspect needs to be taken into consideration: the association can be moved to or from the content of schema class C_S . If the association was moved to $\text{content}(C_S)$ (i.e. $\text{child}(\tilde{R}')$ became a top class) and \tilde{R}' has a name, then each instance of R' can become a basis of a new valid document. This way, if there were more instances of R' in the document D , several adapted documents can be created from D . This adaptation is correct, but the user should always be warned before the algorithm proceeds to perform it, because the consequences can be large-scale (Fig. 10).

If the association is moved from $\text{content}(C_S)$ elsewhere in the tree, another top class must be selected as the new root. Candidates are those top classes that have instances in the adapted documents or classes that serve as wrappers for such classes. Nonetheless, if there is more than one such candidates, it is up to the user to choose, whereas the class selected as the new root can change from document to document. The alternative in cases when more candidates are available is again to produce one adapted document for each such candidate.

classMoved (\tilde{C}', \tilde{R}'_n) *contentModelMoved* (\tilde{M}', \tilde{R}'_n) As opposed to the previous case, it is now the node which is moved, not the association leading to it.

We introduce separate predicates to cover all possible changes between two version of the model, but their treatment during adaptation is analogous. Let $\tilde{N}' = \tilde{C}'/\tilde{M}'$, $N' = \text{getInVer}(\tilde{C}', v)/\text{getInVer}(\tilde{M}', v)$ for *classMoved/contentModelMoved* respectively. The instances of N' will be converted to the instances of \tilde{N}' in the content of \tilde{R}'_n . Again as for *attributeMoved* and *associationMoved*, adaptation can be offered in some particular cases, but for the general case, the user must provide his/her adaptation function.

3.6 Structural representatives changes

srIntroduced (\tilde{C}', \tilde{C}'_r) When a structural representative is introduced, an instance of content inherited from the represented class must be created. However, the motivation for introducing a new structural representative is when a certain part of the schema is factored out for reuse. For instance, when translated to XML Schema language, introducing a structural representative can be translated as adding a new complex type and referencing this type from an element. In that case, the content is already present in the document at the right place (only it was an instance of another subtree) and no adaptation is needed. This means that the adaptation of the *srIntroduced* change can be skipped.

A more general case is when the part of the schema is factored out and referenced from several classes using the structural representative construct, i.e. several instances of the *srIntroduced* change will be detected, all sharing the parameter \tilde{C}'_r . In that case again, if the content is already a part of the document, the adaptation can be skipped.

srCleared (\tilde{C}') Converting structural representative to a regular class requires removing the content that is the instance of the content inherited from the represented class. It is an opposite case to *srIntroduced* and so is the adaptation.

srChanged (\tilde{C}', \tilde{C}'_r) The change of the represented class from one class to another one is not an operation that we expect the user to perform frequently. If it is detected, the adaptation is, in fact, a combination of adaptation of *srRemoved* and *srIntroduced* changes described above.

3.7 Generating content

As mentioned before, certain modifications in the schema may require a new content to be added into some (or all) documents in $\mathcal{T}(S')$ to adapt the documents. This happens in particular when:

- A new mandatory construct is added into the schema, either a class via *classAdded* (C' , s.t. $card(parent(C')) = l..u, l > 0$) or an attribute via *attributeAdded* (A' , s.t. $card(A') = l..u, l > 0$).
- A cardinality interval was extended from $l_1..u_1$ to $l_2..u_2$, where $l_1 < l_2$ (using *attributeCardinalityChanged* or *associationCardinalityChanged*).
- A construct was moved or deleted from a content choice and its instance in the XML document must be replaced by instance of one of the other components in the content choice (using *classMoved*, *contentModelMoved*, or *associationMoved*).

The following text discusses several possible solutions.

Default values of attributes One of the easiest ways is to introduce function $default : S'_a \rightarrow \mathcal{D}^*$ that would assign a default value for PSM attributes. This value could then be used each time an attribute instance needs to be generated. XML schema languages routinely provide constructs for specifying default values of attributes, so the result is always defined. However, such a solution does not produce realistic data.

User-provided content Leaving the issue up to the user to solve it can be very convenient and in some cases the only correct solution. The adaptation script for evolution from version v to \tilde{v} can be generated by the system with some “blanks” left for the user to fill in or take a form of a parameterized script with the values of parameters to be filled in when executing the script. However, the problem is in cases when the values are document-dependent.

Default complex content The adaptation script can also create the missing element content itself. The structure of the internal nodes is given, for values of leaf nodes and attributes the default values for the given type of each PSM attribute (e.g. an empty string for type `xs:string`) can be utilized.⁴ Where a content model of type choice is present, the first component is always selected. For each attribute and association with cardinality $m..n$, the algorithm creates exactly m instances. Such a content will be called *default instance*. The default instance can serve as a skeleton for the user, who can then fill it in with relevant data.

Generalization of version links for sets As we can see from Definition 11, we defined version links as $1 : 1$ mappings. However, situations fitting $1 : N$ or $M : N$ may emerge in real-world evolution scenarios. Examples may be splitting one attribute into a group of

attributes or replacing one group of attributes with another, for example, splitting an attribute `name` into attributes `first-name` and `last-name` and annotating the change with the information that the value of the old attribute is split among the values of the new attributes by a whitespace in the text. Such annotations can be either selected from a certain pre-defined set, or entered by the user in the form of an integrity constraint.

Utilization of PIM links and referencing other data sources When a new PSM construct is added, it can have an interpretation in the PIM. This semantic information can be used to obtain the correct data when content is generated for the new construct.

An example could be the situation when the data is stored in a relational database. The content for a PSM construct can be then retrieved from the data in the database. However, this would be possible only if there is a model of a relational database linked to the PIM. (We discuss this topic in detail in Section 4.) When content is already present in the system (in any form, e.g., in a relational database, another XML documents etc.), it can be retrieved by the adaptation script.

Figure 11 contains a simple example of schema evolution and Fig. 12 shows an adaptation script that references an additional document to provide the necessary data. In particular, a new attribute `email` was added to the schema. The adaptation script is supplied with an additional document `CustomerData.xml`, which contains a list of emails of those customers, for which email addresses are available. Even though the schema is backwards compatible (`email` is optional), the customer record will be enriched by the added content where possible. The adaptation script can be used for all the documents from $\mathcal{T}(S')$.

3.8 Adaptation example

We conclude this section with an example of an adaptation stylesheet generated by the system. As an example, we show the adaptation script in XSL that transforms

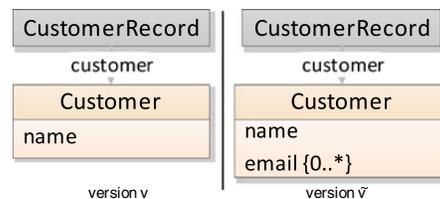


Fig. 11 Example of added content

⁴Another possibility is to utilize function *default* proposed in earlier in this section.

<pre> <?xml version="1.0" encoding="utf-8"?> <xsl:stylesheet version="2.0" xmlns:xsl=".."> <xsl:output method="xml" indent="yes" /> <xsl:variable name="cl" select= "document('CustomerList.xml')/Customers" /> <xsl:template match="/Customer"> <Customer> <xsl:copy-of select="@name" /> <xsl:copy-of select="@customer-no" /> <xsl:variable name="cn" select="@customer-no" /> <xsl:copy-of select= "\$cl/Customer[@customer-no=\$cn]/email" /> </Customer> </xsl:template> </xsl:stylesheet> </pre>	<pre> <?xml version="1.0"?> <Customers> <Customer customer-no='34'> <firstname>John</firstname> <lastname>Doe</lastname> <email>john.doe@example.org</email> <email>john.doe@company.com</email> </Customer> ... <Customer customer-no='77'> <firstname>Martin</firstname> <lastname>Smith</lastname> <email>martin.smith@domain.org</email> </Customer> </Customers> </pre>
(a) Adaptation stylesheet	(b) Referenced document

Fig. 12 Stylesheet and document with additional data

purchase documents processed by the print server. The original and evolved version are depicted in Fig. 13. The following changes were made upon the schema:

1. The print server can no longer access the database, thus the printed purchase documents must contain all the data themselves. That is the reason why purchased items have name attribute.
2. The address is printed on envelopes by a separate envelope printer which accepts a rigid data structure defined by class `EnvelopeType`. Attributes of `EnvelopeType` are “semantically equivalent” to attributes of `Address`.
3. The currency code which was in the original version a part of the value of `price` attribute is now a separate attribute.
4. Attribute `name` was moved from class `Address` to class `Customer`.
5. There is only one note for the whole purchase in the evolved version. Attribute `note` was moved from `Items` to `PurchaseP`. The adaptation script should concatenate the existing notes.

The list of the detected change instances is as follows (for better readability, we use names of classes, attributes and associations in the predicate instances):

$$attributeTypeChanged_{ins} = \{(price, integer)\}$$

$$associationMoved_{ins} = \{(customerAddress, Customer, 0)\}$$

$$attributeMoved_{ins} = \{(name, customer, 1)\}$$

$$attributeIndexChanged_{ins} = \{(Address.number, 0), (Address.streetCode, 1), (Address.cityCode, 2), (Address.zipCode, 3)\}$$

$$classAdded_{ins} = \{(Envelope, EnvelopeType)\}$$

$$attributeAdded_{ins} = \{(currencyCode, item, 3), (salutation, Envelope, 0), \dots^5\}$$

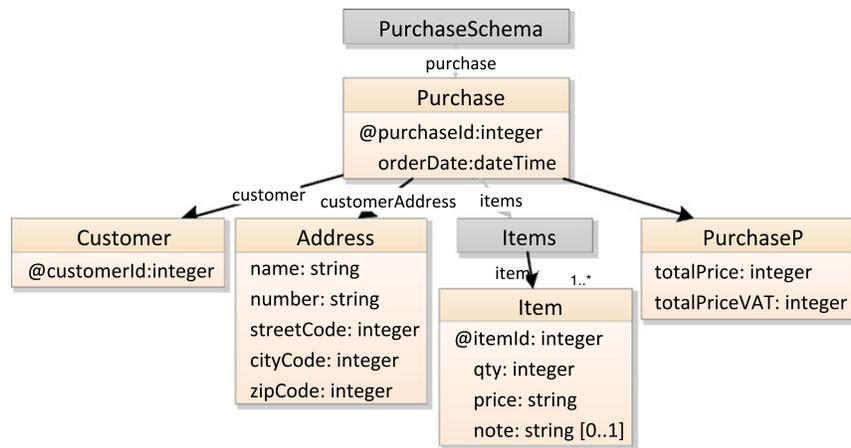
$$attributeMoved_{ins} = \{(note, PurchaseP, 2)\}$$

To demonstrate the possibility of generalized version links (described in the previous subsection), we assume a version link l between two sets of attributes:

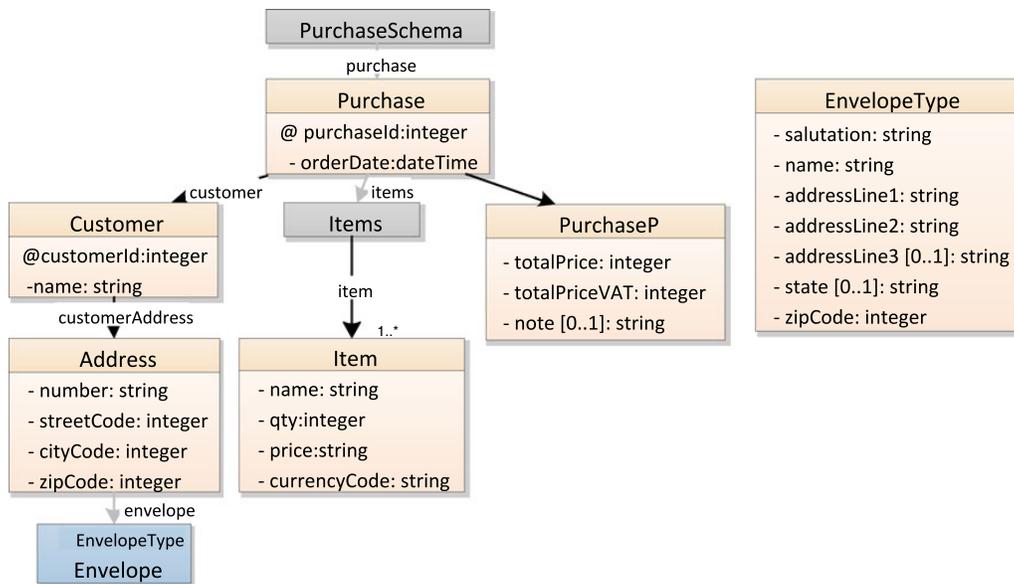
$$l = (\{number_1, streetCode_1, cityCode_1, zipCode_1\}, \{addressLine1-3_2, state_2, zipCode_2\})$$

The generated adaptation XSLT script is listed in Fig. 14. It follows the structure of the input document. It consists of templates matching the elements in the input document (`purchase`, `customer`, etc.) and named templates for the elements that are

⁵And other attributes of class `EnvelopeType`.



(a) Original purchase schema



(b) Evolved purchase schema

Fig. 13 Two versions of a purchase schema

added in the new version (*envelope*, *note*). Template *customer* adapts to the migratory change *associationMoved* by forcing the processor to progress to *customerAddress* element. The adaptation of *price* and the added attribute *currencyCode* is achieved using user-provided functions. For the adaptation of *notes*, the user selected a general ‘concat’ adaptation (since the situation corresponds to the case of adaptation of *attributeMoved*, see p. 21). Element *envelope* is filled using information from the link *l* (see template named *envelope*). In this template, also the data from external source is required to fill in the variable *customerRec* (the fetched data is used

to decide, whether the customer is male or female). Similarly for variable *itemRec* in the template for *items* (here the variable is used to provide value for *name* attribute). Two last templates are used to process nodes that were not changed.

3.9 Implementation

We gradually incorporate the results of all our research activities related to our 5-level framework (see Fig. 1) into an experimental tool *eXolutio* (Klímek et al. 2011). The modules related to this paper are *versioning* (which allows to keep and design multiple version of the

Fig. 14 Adaptation script for the schemas depicted in Fig. 13

```

<xsl:stylesheet>
  <xsl:template match='purchase'>
    <purchase>
      <xsl:apply-templates select='@purchaseId' />
      <xsl:apply-templates select=
        'orderDate | customer | items | totalPrice | totalPriceVAT' />
      <xsl:call-template name='note' />
    </purchase>
  </xsl:template>

  <xsl:template match='customer'>
    <customer>
      <xsl:copy-of select='@customerId' />
      <xsl:apply-templates select='../customerAddress/name' />
      <xsl:apply-templates select='../customerAddress' />
    </customer>
  </xsl:template>

  <xsl:template match='customerAddress'>
    <address>
      <xsl:apply-templates select='number | streetCode | cityCode | zipCode' />
      <xsl:call-template name='envelope' />
    </address>
  </xsl:template>

  <xsl:template match='item'>
    <xsl:variable name='itemRec'>...</xsl:variable>
    <item>
      <xsl:apply-templates select='@*' />
      <name><xsl:value-of select='$itemRec/name' /></name>
      <xsl:apply-templates select='qty | price' />
      <xsl:call-template name='currencyCode' />
    </item>
  </xsl:template>

  <xsl:template name='envelope'>
    <xsl:variable name='customerRec'>...</xsl:variable>
    <envelope>
      <salutation>
        <xsl:value-of select='if ($customerRec/sex eq male) then "Sir" else "Madam"' />
      </salutation>
      <name><xsl:value-of select='name' /></name>
      <addressLine1>
        <xsl:value-of select='concat(number, " ", cityCode)' />
      </addressLine1>
      <addressLine2><xsl:value-of select='cityCode' /></addressLine2>
      <zipCode><xsl:value-of select='zipCode' /></zipCode>
    </envelope>
  </xsl:template>

  <xsl:template match='price'>
    <price><xsl:value-of select='number(substring-before(., " "))' /></price>
  </xsl:template>

  <xsl:template name='currencyCode'>
    <currencyCode><xsl:value-of select='substring-after(price, " ")' /></currencyCode>
  </xsl:template>

  <xsl:template name='note'>
    <note><xsl:value-of select='user:concatNote(items/item/note)' /></note>
  </xsl:template>

  <xsl:function name='user:concatNote' >
    <xsl:param name='notes' />
    <xsl:for-each select='$notes'>
      <xsl:value-of select='concat(..@itemId, ': ', .., '; ')' />
    </xsl:for-each>
  </xsl:function>

  <!-- pass-through nodes -->
  <xsl:template match='items'>
    <xsl:copy><xsl:copy-of select='@*' /><xsl:apply-templates select='*' /></xsl:copy>
  </xsl:template>
  <!-- plain-copy nodes -->
  <xsl:template match=
    '@purchaseId | orderDate | @customerId | name | qty | number | streetCode |
    cityCode | zipCode | @itemId | totalPrice | totalPriceVAT' >
    <xsl:copy-of select='.' />
  </xsl:template>
</xsl:stylesheet>

```

system in one project) and *adaptation* (which generates the adaptation stylesheet as described in Section 3). Both work tightly with the *evolution* module, which deals with propagating changes to all levels (see Nečaský et al. (2011a) for more details).

We chose XSLT as an implementation language. We were motivated by the wide support for XSLT both in native XML databases and in XML-enabled relational databases. The generated stylesheet, when applied at the adapted document, acts as the *adapt* function. The

description of the process of generating templates can be found in Malý et al. (2011).

4 Open problems

Even though our approach is complex and covers the most common XML schema features, operations and cases, there is still a space of possible improvements. They are mostly related to more advanced constructs and more precise means of specification of data structure, or to operations with the data.

Integrity constraints The UML allows the designer to specify constraints and invariants in the model via OCL in those situations, where classes and associations do not describe the model sufficiently. At the level of XML schemas, constraints are required too. And some types of constraints are impossible to be defined via languages based on RTGs, i.e. DTD and XML Schema. Examples of such constraints can be choices between groups of attributes or so-called *co-occurrence constraints* (e.g. element E_1 must occur only if the value of element E_2 is v_2). To allow for such constraints, XML Schema was extended with the possibility to declare non-RTG constructs `assert` and `report` inspired by key constructs Schematron language.

Since we modified UML to serve us in XML modeling, we plan to modify OCL to serve us to define constraints in XML schemas. Likewise PSM schemas can be translated to XSDs, we will be able to translate the PSM-level constraints to Schematron schemas analogously.

From the evolution point of view, with OCL constraints, it will be possible to track changes in semantics. For example, consider the situation when the request for customer history returns the list of all purchases in the old version, but in the new version, the list should contain only realized purchases. The structure of the schema will remain unchanged, but in the new version, a new constraint will be added. The evolution algorithm will be able to adapt the document accordingly via deleting all the unrealized purchases. Since all the existing evolution frameworks only deal with structure and do not recognize semantics, none of them is capable at least to detect such a change.

Advanced constructs UML class diagrams define other constructs apart from the fundamental ones that we adopted for the PIM level. Besides compositions, aggregations and association classes, the main contribution to the approach will be the notion of inheritance. Defined at the PIM level, it can be transferred to the

PSM level and provide an alternative way of content reuse to the means of the structural representatives.

Version links for imported schemas As we have mentioned, before our algorithm can be used to work with imported schemas, it requires the relation $\mathcal{V}\mathcal{L}$ to be defined (i.e. version links joining the previous and the evolved version of each construct). This relation can be created manually by the user, but it is a time-consuming process. It would be useful to extend the system with a heuristic that could create a larger part of $\mathcal{V}\mathcal{L}$ automatically and ask for user input only in the unresolved cases. There exists a lot of methods for finding similarities and patterns among two XML schemas (a survey can be found in Nečaský and Mlýnková (2009)). Outcomes of these methods can be transferred into finding similarities between two PSM schemas.

Generating content To date, our approach is able to deal with changes that modify the structure and data present in the document. In Section 3.7 we indicated approaches how to handle adding content during adaptation. Besides the basic methods of using default values or letting the user to supply the content manually, other methods can be utilized. When the required data is already present in the system, but not in the adapted documents themselves (e.g. in a database, in other XML documents, available through some service interface), these sources can be used to retrieve the data during adaptation. To achieve this, the data should be mapped to the PIM. If the mapping is designed appropriately, it will be possible to reconstruct the appropriate queries/service calls. The capabilities of this approach can be further enhanced when support for the integrity constraints is added to the system.

Operations with the data The last but not least related open problem is the propagation of changes from data structures to respective operations, i.e. XPath/XQuery queries, XSLT scripts etc. Surprisingly, the amount of related works is relatively low. However, though the problem seems to be completely different, it may be solved in a similar way (Polák 2011). We may define a model of the respective query language and its mapping to the model of the data. Then, we need to specify respective edit operations and their propagation. The problem is that even the simplest XML query language (XPath) is much richer than the language for description of data structures, so there are many cases to be covered, not all of them can be solved manually and some of them have multiple solutions when user interaction is required. However, even a respective detailed case study is still missing.

5 Related work

Currently there exists a significant number of approaches that detect changes between two schemas of data and output the sequence of edit operations that enable their re-validation (Malý et al. 2011). There are two possible ways to recognize changes—*recording the changes* as they are conducted during the edit process or *comparing two versions of the schema*.

The edit operations can be also variously classified. For instance, Tan and Goh (2005) proposes *migratory* (e.g. movements of elements/attributes), *structural* (e.g. adding/removal of elements/attributes) and *sedentary* (e.g. modifications of simple data types) operations. Classification according to complexity distinguishes *atomic* and *composite* operations.

The changes can also be expressed variously and more or less formally. For instance in Cavalieri (2010) a language called *XUpdate* is described. In Guerrini and Mesiti (2009) the authors propose the *XSchemaUpdate language*.⁶

5.1 Version comparison

If we analyze the two types of approaches in the area of XML data and XML schemas, we can find numerous methods that focus on the latter class, since change detection of two given versions of data is a key part of, e.g., data integration, versioning, similarity evaluation, etc. Rahm and Bernstein (2001) In all the cases we are interested in the sequence of edit operations, which is further used for mapping purposes, evaluation of the degree of difference etc. At the level of XML documents we can restrict ourselves to detecting changes between trees, either ordered (Cobena et al. 2002; Leonardi and Bhowmick 2007) or unordered (Chawathe and Garcia-Molina 1997; Wang et al. 2003). Since the problem is proven to be NP-hard (Cobena et al. 2002), various heuristics reflecting the respective application are often incorporated, as well as optimization strategies for processing large documents and gaining better results using relational databases (Leonardi and Bhowmick 2006, 2007), XPath queries (Qeli et al. 2006), tuning steps (Lee and Kim 2006) etc.

In the area of XML schemas the amount of approaches is much lower. It is given mainly by the fact that in this case we do not compare two trees, but two general graphs possibly with cycles and with nodes of highly different semantics (elements, attributes, operators). One of the first approaches that deals with

detection of changes between two given DTDs (possibly extensible to XSDs) which shows that the approaches for XML data cannot be directly applied for XML schemas due to the mentioned semantics of nodes is proposed in Leonardi et al. (2007). It exploits a heuristics based on MD5 hashing. There exists also an approach that detects changes among XML schemas for the purpose of evaluation of their similarity on the basis of classical tree-edit distance (Wojnar et al. 2010). On the other hand, the algorithm in Kwietniewski et al. (2010) starts at roots of the source and target XML schema and continues recursively (the routine attempts to match sets of children of two already matched nodes). The mapping is “best-effort” and partial, hence the produced XSLT script does not guarantee correct revalidation (the output document will not be valid against the target schema)—the user is expected to adjust it. The commercial tool (Altova/DiffDog <http://www.altova.com/diffdog/>) offers semi-automatic schema comparison (of XML schemas, i.e. at the logical level) and subsequent creation of an adaptation script, but the task of resolving ambiguities inherent in schema comparison approaches is left up to the user.

5.2 Recording changes

A system that records changes has the advantage of knowing the sequence of operations that were performed. However, the sequence does not have to be optimal, because the user could reach the result using various more or less reasonable sequences of operations. The user must also be provided with a user friendly interface; hence, it is quite common that the changes are expressed in a visualization or model of the schema. In general, the user works with a kind of evolution system and often the changes are propagated directly one-by-one, i.e. an *incremental (immediate) re-validation* is used.

System *X-Evolution* (Guerrini et al. 2007) is built upon a graphical editor for creating schemas in the XML Schema language. Each single evolution operation executed upon the schema is immediately propagated to valid documents, whereas backwards compatible operations are recognized. System *XEM (XML Evolution Management)* (Su et al. 2002) has a similar strategy, i.e. immediate revalidation, but supports changes both in both DTD and XML documents represented as directed acyclic graphs. The set of supported operations is proven to be sound and complete. But, since renaming and moving is not supported (only adding and deleting), when the operations are propagated, they lead to removing and re-creating of significant parts of the data. Similarly, there are

⁶Not to be confused with XQuery Update Facility (W3C 2011)

papers which deal with propagation of a single change expressed in DTD (Al-Jadir and El-Moukaddem 2003; Coox 2003) or XSD (Tan and Goh 2005; Cavalieri 2010) to respective XML documents. There also exists an opposite approach that enables one to evolve XML documents and propagate the changes to their XML schema (Bouchou et al. 2004).

System *CoDEX* (*Conceptual Design and Evolution of XML Schemas*) (Klettke 2007) is an example of an approach to schema evolution using the true recording approach. The changes made in the visualization of the schema are logged and when the evolution process is finished, the resulting sequence of changes is normalized and then performed in the XML schema and respective XML documents. Moves and renames are supported, but the tool does not support multiple instances of schema constructs. They show an example why this can lead to unexpected results as described by the example in Section 1. The reason for this is that the model used by CoDEX is not a conceptual model, even though it hides some technical details of XML schema languages.

In Dominguez et al. (2011) the authors propose an approach for expressing changes at the level of UML classes and their propagation to respective XML schemas and XML documents with the emphasis on *traceability*, i.e. preserving the links between the levels. This is probably the closest system to our basic idea of five-level evolution framework (Nečaský and Mlýnková 2009a); however, the authors consider only its part. Considering the aim of this paper, i.e. propagation of changes from XML schemas to XML documents, the authors of Dominguez et al. (2011) do not consider operation move or other more complex operations (adding, deleting, renaming and changing a property to a class is supported) and the framework provides directly the output documents, not the set of operations in some standard syntax, that might be further processed with regard to the respective operation.

In Raghavachari and Shmueli (2007) the authors propose an algorithm for *incremental schema validation*, i.e. checking validity of an XML document with regard to a modified schema, whereas the aim is not to check the whole document, but only necessary parts. For this purpose, the old and the new version of the schema are analyzed and an *intersection automation* is built from the two versions of the schemas. A modification of the algorithm also enables to efficiently check validity of a modified XML document against the new version of XML schema.

As we can see, even though the amount of approaches and interesting contributions is high, there are still many drawbacks and open issues. In this paper we show

that the basic features of our framework (namely conceptual modeling levels and preserving respective relations) bring many optimizations and enable to preserve not only structural, but also semantic fidelity. Since the very first design we have also focussed on the indicated problems of selection of the set of edit operations and output formats and, hence, we can avoid them.

6 Conclusion

The aim of our work was to propose an approach to XML schema evolution built upon a conceptual model for XML schemas. It identifies changes in the schema, determines the impact of the changes on the existing documents valid against its old version and produces an adaptation script when adaptation of the existing documents is necessary with regard to the new version.

The key contributions of the approach can be summed up as follows:

- We exploit the idea of a conceptual model of XML schemas and, hence, the user is provided with a user-friendly tool for expressing changes. We have built the algorithm upon a general schema model, which is proven (Nečaský et al. 2011b) to cover DTD, XML Schema and RELAX NG.
- Our approach is integrated within a five-level evolution framework where the two conceptual levels—platform-independent and platform-specific—together with the versioning support enable to model multiple schema versions at once.
- We overcame the problem inherent in all approaches comparing/mapping two versions of schemas via introducing the version links. Each construct in the model is then correctly connected with its counterpart constructs in all other versions, where the construct exists. Adding the version links allowed us to define changes that can occur between two versions v and \tilde{v} of a schema and an algorithm that detects these changes.
- Our approach outputs an XSLT script that adapts the modified XML documents with regard to a new version of a schema. The adapted document preserves semantical meaning of the constructs due to utilizing the version links.
- The user works with and evolves a conceptual model of a schema and the mappings are defined for the conceptual model as well. This is a significant improvement compared to working directly with often lengthy and hard to read XML schemas.

Our approach is efficient, because the phase of adaptation is clearly separated from schema evolution. We

do not require to adapt after each change in a schema, but only when the whole schema is evolved (and all evolution steps are consolidated). We are able to decide, whether changes made in the schema are backwards-compatible or not. The generated adaptation script is one-pass and skips those portions of adapted documents that were not changed (see Malý et al. (2011) for details).

The algorithm in its current version deals with revalidation of (1) structure and (2) data already present in the document. Since new data are often required for new versions, we will focus in our future work on obtaining this data for the revalidated documents. For this purpose, we will utilize the existing connection between PIM and PSM and a new similar connection between PIM and the model of a data storage (e.g. an ER schema (Thalheim 2000)).

Finally, a complex system, besides a precise definition of how the data are structured, requires a support for modeling and checking integrity constraints (ICs). ICs also change as system evolves and can also be used to describe evolution operations in greater detail. Support for ICs will further enhance capabilities and applicability of the algorithm.

In our future work we will also consider the operational level expressed in, e.g., XQuery (W3C 2010b) or XSLT (Kay 2007). The goal is to extend the *adapt* function introduced in this paper also to queries expressed in various query languages. Having a set of queries working on a set of XML documents valid with respect to an old version of a schema, the extended *adapt* function would adapt the queries so that they correctly work on the XML documents adapted w.r.t. to a new version of the schema.

References

- Al-Jadir, L., & El-Moukaddem, F. (2003). Once upon a time a DTD evolved into another DTD. In *Object-oriented information systems* (pp. 3–17). Berlin: Springer.
- Bouchou, B., Duarte, D., Alves, M.H.F., Laurent, D., Musicante, M.A. (2004). Schema evolution for XML: A consistency-preserving approach. In *Mathematical foundations of computer science* (pp. 876–888). Prague: Springer.
- Cavaliere, F. (2010). EXup: An engine for the evolution of XML schemas and associated documents. In *EDBT '10: Proc. of the 2010 EDBT/ICDT workshops* (pp. 1–10). New York: ACM.
- Chawathe, S.S., & Garcia-Molina, H. (1997). Meaningful change detection in structured data. In J. Peckham (Ed.), *SIGMOD conference* (pp. 26–37). ACM Press.
- Clark, J., & Makoto, M. (2001). RELAX NG specification. Oasis. <http://www.oasis-open.org/committees/relax-ng/spec-20011203.html>. Accessed 22 Nov 2011.
- Cobena, G., Abiteboul, S., Marian, A. (2002). Detecting changes in XML documents. In *ICDE* (pp. 41–52). IEEE Computer Society.
- Coox, S.V. (2003). Axiomatization of the evolution of XML database schema. *Programming and Computer Software*, 29(3), 140–146.
- Dominguez, E., Lloret, J., Pérez, B., Rodríguez, Á., Rubio, Á.L., Zapata, M.A. (2011). Evolution of XML schemas and documents from stereotyped UML class models: A traceable approach. *Information and Software Technology*, 53, 34–50.
- Guerrini, G., & Mesiti, M. (2009). XML schema evolution and versioning: Current approaches and future trends. In E. Pardede (Ed.), *Open and novel issues in XML database applications: Future directions and advanced technologies* (pp. 66–87). Idea Group Publishing.
- Guerrini, G., Mesiti, M., Sorrenti, M.A. (2007). XML schema evolution: Incremental validation and efficient document adaptation. In D. Barbosa, A. Bonifati, Z. Bellahsene, E. Hunt, R. Unland (Eds.), *Database and XML technologies, Lecture notes in computer science* (Vol. 4704, pp. 92–106). Berlin/Heidelberg: Springer. doi:10.1007/978-3-540-75288-2_8.
- ISO (2005). Information Technology Document Schema Definition Languages (DSDL) Part 3: Rule-based Validation Schematron. ISO/IEC 19757-3.
- ISO (2008). ISO/IEC 9075-14:2008—SQL—Part 14: XML-Related Specifications (SQL/XML). http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=45499. Accessed 22 Nov 2011.
- Kay, M. (2007). XSL transformations (XSLT) version 2.0. W3C. <http://www.w3.org/TR/xslt20/>. Accessed 22 Nov 2011.
- Klettke, M. (2007). Conceptual XML schema evolution—The CoDEX approach for design and redesign. In *Workshop proceedings datenbanksysteme in Business, Technologie und Web (BTW 2007)* (pp. 53–63). Aachen, Germany.
- Klímeček, J., Malý, J., Nečáský, M. (2011). eXolutio—A tool for XML data evolution. <http://exolutio.com>. Accessed 22 Nov 2011.
- Klímeček, J., & Nečáský, M. (2010). Semi-automatic integration of web service interfaces. In *IEEE international conference on web services* (pp. 307–314).
- Kwiatkiewski, M., Gryz, J., Hazlewood, S., Van Run, P. (2010). Transforming XML documents as schemas evolve. *Proceedings of the VLDB Endowment*, 3(1–2), 1577–1580. <http://dl.acm.org/citation.cfm?id=1920841.1921043>.
- Lee, S., & Kim, D. (2006). X-tree diff+: Efficient change detection algorithm in XML documents. In E. Sha, S.K. Han, C.Z. Xu, M.H. Kim, L. Yang, B. Xiao (Eds.), *Embedded and ubiquitous computing. Lecture notes in computer science* (Vol. 4096, pp. 1037–1046). Berlin: Springer.
- Leonardi, E., & Bhowmick, S.S. (2006). Xandy: A scalable change detection technique for ordered XML documents using relational databases. *Data & Knowledge Engineering*, 59(2), 476–507.
- Leonardi, E., & Bhowmick, S.S. (2007). XANADUE: A system for detecting changes to XML data in tree-unaware relational databases. In *Proceedings of the 2007 ACM SIGMOD international conference on management of data, SIGMOD '07* (pp. 1137–1140). New York: ACM.
- Leonardi, E., Hoai, T.T., Bhowmick, S.S., Madria, S.K. (2007). DTD-diff: A change detection algorithm for DTDs. *Data & Knowledge Engineering*, 61(2), 384–402.
- Malý, J., Mlýnková, I., Nečáský, M. (2011). *On XML document transformations as schema evolves—A survey of current approaches*. ISD 2010.

- Malý, J., Mlýnková, I., Nečaský, M. (2011). XML data transformations as schema evolves. In *ADBIS '11: Proc. of the 15th advances in databases and information systems*. Vienna: Springer.
- Miller, J., & Mukerji, J. (2003). MDA guide version 1.0.1. Object management group. <http://www.omg.org/docs/omg/03-06-01.pdf>.
- Murata, M., Lee, D., Mani, M., Kawaguchi, K. (2005). Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4), 660–704.
- Nečaský, M., & Mlýnková, I. (2009). Exploitation of similarity and pattern matching in XML technologies. In *DATESO 2009, CEUR workshop proceedings* (Vol. 471, pp. 90–104). Matfyz Press.
- Nečaský, M. (2009). Conceptual modeling for XML. *Dissertations in database and information systems* (Vol. 99). Amsterdam: IOS Press.
- Nečaský, M. (2009). Reverse engineering of XML schemas to conceptual diagrams. In *Proceedings of the 6th Asia-Pacific conference on conceptual modelling* (pp. 117–128). Wellington: Australian Computer Society.
- Nečaský, M., & Mlýnková, I. (2009a). Five-level multi-application schema evolution. In *DATESO '09* (pp. 90–104).
- Nečaský, M., & Mlýnková, I. (2009b). On different perspectives of XML schema evolution. In *FlexDBIST'09: Proceedings of the 5th international workshop on flexible database and information system technology*. Linz: IEEE Computer Society.
- Nečaský, M., & Mlýnková, I. (2010). A framework for efficient design, maintaining, and evolution of a system of XML applications. In *Proceedings of the Databases, Texts, Specifications, and Objects, DATESO '10* (pp. 38–49). Matfyz Press.
- Nečaský, M., Klímek, J., Malý, J., Mlýnková, I. (2011a). Evolution and change management of XML-based systems. *Journal of Systems and Software*. doi:10.1016/j.jss.2011.09.038. <http://www.sciencedirect.com/science/article/pii/S0164121211002524>.
- Nečaský, M., Mlýnková, I., Klímek, J., Malý, J. (2011b). When conceptual model meets grammar: A dual approach to XML data modeling. *Data & Knowledge Engineering*. doi:10.1016/j.datak.2011.09.002. <http://www.sciencedirect.com/science/article/pii/S0169023X1100125X>.
- Object Management Group (2007a). UML infrastructure specification 2.1.2. <http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF>. Accessed 28 Feb 2012.
- Object Management Group (2007b). UML superstructure specification 2.1.2. <http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF>. Accessed 28 Feb 2012.
- Biron, P.V., & Permanente, K.A.M. (2004). *XML schema part 2: Datatypes* (2nd Edn.). W3C <http://www.w3.org/TR/xmlschema-2/>. Accessed 22 Nov 2011.
- Polák, M. (2011). XML query adaptation. Master Thesis, Charles University in Prague, Czech Republic. <http://www.ksi.mff.cuni.cz/~mlynkova/dp/Polak.pdf>. Accessed 22 Nov 2011.
- Qeli, E., Gllavata, J., Freisleben, B. (2006). Customizable detection of changes for XML documents using XPath expressions. In D.C.A. Bulterman, D.F. Brailsford (Eds.), *Proceedings of the 2006 ACM symposium on document engineering* (pp. 88–90). Amsterdam: ACM Press.
- Raghavachari, M., & Shmueli, O. (2007). Efficient revalidation of XML documents. *IEEE Transactions on Knowledge and Data Engineering*, 19, 554–567. doi:10.1109/TKDE.2007.1004. <http://dl.acm.org/citation.cfm?id=1263133.1263349>.
- Rahm, E., & Bernstein, P.A.: (2001). A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4), 334–350.
- Su, H., Kramer, D.K., Rundensteiner, E.A. (2002). *XEM: XML evolution management*. Tech. Rep. WPI-CS-TR-02-09, Computer Science Department, Worcester Polytechnic Institute, Worcester, Massachusetts.
- Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F. (2008). *Extensible Markup Language (XML) 1.0* (5th edn.). W3C <http://www.w3.org/TR/REC-xml/>.
- Tan, M., & Goh, A. (2005). Keeping pace with evolving XML-based specifications. In *EDBT'04 workshops* (pp. 280–288). Berlin: Springer.
- Thalheim, B. (2000). *Entity-relationship modeling: Foundations of database technology*. Berlin: Springer.
- Thompson, H.S., Beech, D., Maloney, M., Mendelsohn, N. (2004). XML schema part 1: Structures (2nd edn.). W3C <http://www.w3.org/TR/xmlschema-1/>.
- W3C (2004). Document Object Model (DOM) specification. <http://www.w3.org/DOM/>. Accessed 22 Nov 2011.
- W3C (2010a). XML Path Language (XPath) 2.0. <http://www.w3.org/TR/xpath20/>. Accessed 22 Nov 2011.
- W3C (2010b). XQuery 1.0: An XML query language. <http://www.w3.org/TR/xquery/>. Accessed 22 Nov 2011.
- W3C (2011). XQuery update facility 1.0 specification. <http://www.w3.org/TR/xquery-update-10/>. Accessed 22 Nov 2011.
- Wang, Y., DeWitt, D.J., Cai, J.Y. (2003). X-diff: An effective change detection algorithm for XML documents. In *International conference on data engineering* (p. 519).
- Wojnar, A., Mlýnková, I., Dokulil, J. (2010). Structural and semantic aspects of similarity of document type definitions and XML schemas. *Information Sciences*, 180(10), 1817–1836. Special Issue on Intelligent Distributed Information Systems.

Jakub Malý received his Master's degree in Computer Science in June 2010 from the Charles University in Prague, Czech Republic, where he currently is a Ph.D. student at the Department of Software Engineering. His research areas involve conceptual modeling of XML data and evolution of XML applications, integrity constraints in models and Object Constraint Language. He has published two journal and eight conference papers.

Martin Nečaský received his Ph.D. degree in Computer Science in 2008 from the Charles University in Prague, Czech Republic, where he currently works at the Department of Software Engineering as an assistant professor. He is an external member of the Department of Computer Science and Engineering of the Faculty of Electrical Engineering, Czech Technical University in Prague. His research areas involve XML and Linked data design, integration and evolution. He is an organizer/PC chair/member of more than 10 international events. He has published more than 40 papers (two received Best Paper Award). He has published three book chapters and a book.

Irena Mlýnková received her Ph.D. degree in Computer Science in 2007 from the Charles University in Prague, Czech Republic. She is an assistant professor at the Department of Software Engineering of the Charles University and an external member of the Department of Computer Science and Engineering of the Czech Technical University. She has published more than 60 papers and books on management of XML data, structural similarity, analysis of real-world data, synthesis of XML data, XML benchmarking, XML schema inference and application evolution. Four gained the Best Paper Awards. She is a PC member or reviewer of 15 international events and co-organizer of three international workshops.

Chapter 4

Heuristic Methods for Inference of XML Schemas: Lessons Learned and Open Issues

Irena Mlýnková
Martin Nečaský

Published in *Informatica – the International Journal*. IOS Press, 2012. ISSN
0868-4952. (in press)

Impact Factor: 1.627
5-Year Impact Factor: 1.074



HEURISTIC METHODS FOR INFERENCE OF XML SCHEMAS: LESSONS LEARNED AND OPEN ISSUES *

Irena MLÝNKOVÁ

Department of Software Engineering, Faculty of Mathematics and Physics,
Charles University in Prague, Czech Republic
E-mail: mlynkova@ksi.mff.cuni.cz

Martin NEČASKÝ

Department of Software Engineering, Faculty of Mathematics and Physics,
Charles University in Prague, Czech Republic
E-mail: necasky@ksi.mff.cuni.cz

Abstract. In this paper we focus on a specific class of XML schema inference approaches – so-called *heuristic approaches*. Contrary to *grammar-inferring* approaches, their result does not belong to any specific class of grammars and, hence, we cannot say anything about their features from the point of view of theory of languages. However, the heuristic approaches still form a wider and more popular set of approaches due to natural and user-friendly strategies. We describe a general framework of the inference algorithms and we show how its particular phases can be further enhanced and optimized to get more reasonable and realistic output. The aim of the paper is (1) to provide a general overview of the heuristic inference process and existing approaches, (2) to sum up the improvements and optimizations we have proposed so far in our research group, and (3) to discuss possible extensions and open problems which need to be solved. Hence, it enables the reader to get acquainted with the field fast.

Key words: XML Schema inference, regular-tree grammars, heuristics, integrity constraints;

1. Introduction Without any doubt the XML Bray *et al.* (2008) is currently a de-facto standard for data representation. Its popularity is given by the fact that it is well-defined, easy-to-use and, at the same time, enough powerful. To enable users to specify own allowed structure of XML documents, so-called *XML schema*, the W3C has proposed two languages – DTD Bray *et al.* (2008) and XML Schema Thompson *et al.* (2004); Biron and Malhotra (2004). The former one is directly a part of XML specification and due to its simplicity it is one of the most popular formats for schema specification. The latter language was proposed later, in reaction to the lack of constructs of DTD. The key emphasis is put on simple types, object-oriented features (such as user-defined data types, inheritance, substitutability etc.) and reusability of parts of a schema or whole schemas.

On the other hand, statistical analyses of real-world XML data show that a significant portion of XML documents (52% Mignet *et al.* (2003) of randomly crawled or 7.4% Mlýnková *et al.* (2006) of semi-automatically collected) still have no schema at all. What

*This work was partially supported by the Czech Science Foundation (GAČR), grant number P202/10/0573.

is more, XML Schema definitions (XSDs) are used even less (only for 0.09% Mignet *et al.* (2003) of randomly crawled or 38% Mlýnková *et al.* (2006) of semi-automatically collected XML documents) and even if they are used, they often (in 85% of cases Bex *et al.* (2004)) define so-called *local tree grammars* Murata *et al.* (2005), i.e. grammars that can be defined using DTD as well.

Consequently a new research area of *automatic inference of an XML schema* has opened. The key aim is to create an XML schema for the given sample set of XML documents that is neither too general, nor too restrictive. It means that the set of document instances of the inferred schema is not too broad in comparison with the sample data but, also, it is not equivalent to it. Currently, there are several proposals of respective algorithms, but there is also still a space for further improvements. In particular, since according to the Gold's theorem Gold (1967) regular languages are not identifiable only from positive examples (i.e. sample XML documents expected to be valid against the resulting schema), the existing methods need to exploit either heuristics or a restriction to an *identifiable* subclass of regular languages.

Contributions In this paper we focus on a specific class of XML schema inference approaches – so-called *heuristic approaches*. Contrary to *grammar-inferring* approaches, their result does not belong to any specific class of grammars and, hence, we cannot say anything about their features from the point of view of theory of languages. However, the heuristic approaches still form a wider and more popular set of approaches due to natural and user-friendly strategies. We describe a general framework of the inference algorithms and we show how its particular phases can be further enhanced and optimized to get more reasonable and realistic output. The aim of the paper is (1) to provide a general overview of the heuristic inference process and existing approaches, (2) to sum up the improvements and optimizations we have proposed so far in our research group, and (3) to discuss possible extensions and open problems which need to be solved. Hence, it enables the reader to get acquainted with the field fast..

Outline The rest of the paper is structured as follows: In Section 2 we provide a brief overview of existing XML schema languages and in Section 3 we introduce a formal view of XML schemas. In Section 4 we first describe a general overview of typical phases of XML schema inference algorithm and then analyze particular phases from the point of view of current and well as our improvements. In Section 5 we discuss the remaining open issues to be solved in the area of XML schema inference in general. Finally, in Section 6 we conclude and outline possible future work.

Relation to Previous Papers In this paper we combine and, in particular, extend our several previous results. In paper Mlýnková (2008a) we have provided a brief general overview of both heuristic and grammar-inferring approaches and stated several open issues. In this paper we extend and update it with new findings and conclusions. In papers Vošta *et al.* (2008) and Vyhnánovská and Mlýnková (2010) we provided two approaches that enable one to infer advanced XML Schema constructs that were not supported in previous works; in the latter one also with the usage of user interaction. In our next work we focussed mainly on further possible inputs of the approaches – i.e. XML queries over the data Nečaský and Mlýnková (2009) and an obsolete schema Mlýnková (2009). And, fi-

nally, in paper Mlýnková and Nečaský (2009) we aimed at refining the resulting schemas on the basis of more detailed analysis of data. In this paper we describe the approaches in the context of a common algorithm framework, in more detail and in relation to current works. Our aim is to sum up our as well as general results and provide a detailed and concise overview and summary useful for other researchers in this field.

2. XML Schema Languages Nowadays, there exist several languages for description of an XML schema, i.e. the allowed structure of XML documents. The best known and most commonly used representatives are DTD, XML Schema, RELAX NG, and Schematron.

DTD The simplest and most popular language for description of the allowed structure of XML documents is currently the *Document Type Definition (DTD)* Bray *et al.* (2008). It enables one to specify allowed elements, attributes and their mutual relationships, order and number of occurrences of subelements (using operators ‘,’ ‘|’, ‘?’, ‘+’ and ‘*’), data types (ID, IDREF, IDREFS, CDATA or PCDATA) and allowed occurrences of attributes (IMPLIED, REQUIRED or FIXED). A simple example of a database of employees is depicted in Figure 1.

At first glance it seems that the specification of the allowed structure is sufficient. Nevertheless, even in this simple example we can find several problems. For instance, we are not able to specify the correct structure of an e-mail address. Similarly, we cannot simply specify that a person can have four e-mail addresses at maximum. And, as we can see, the fact that the order of elements `first` and `surname` is not significant cannot be expressed simply as well.

XML Schema With regard to the insufficiency of DTD, the W3C² proposed a more powerful tool – *XML Schema* Thompson *et al.* (2004); Biron and Malhotra (2004) and its instances called *XML Schema Definitions (XSDs)*. An example of an XSD equivalent to the example of a DTD in Figure 1 is depicted in Figure 4. XML Schema involves all the DTD constructs, only the syntax is different (e.g. element `sequence` represents operator ‘,’, `choice` represents ‘|’, content models are specified using `simpleTypes` and `complexType`). It also adds several new constructs that can be divided into “syntactic sugar” and true new constructs extending the expressive power. The former class involves, e.g., precise occurrence ranges (i.e. attributes `minOccurs` and `maxOccurs`) or globally defined items (i.e. `simple/complexType`, `element/attribute`, `groups of elements/attributes`). In the latter class we can find a new rich set of simple data types (such as `integer` or `date`), user-defined simple types, derivation of complex data types from existing ones, advanced identity constraints (i.e. `unique`, `key`, `keyref`) or assertions (i.e. `assert`, `report`) supported since version 1.1 Gao *et al.* (2009); Peterson *et al.* (2009). In addition, each XSD is an XML document, hence, for its processing we can exploit any XML technology.

On the other hand, XML Schema has also several disadvantages. Firstly, as we can see from the examples in Figure 1 and 4, the description of an XSD is much longer and less

²<http://www.w3.org/>

```

<!ELEMENT employees (person)+>
<!ELEMENT person (name, email, relationships?)>
<!ATTLIST person id ID #REQUIRED>
<!ATTLIST person note CDATA #IMPLIED>
<!ATTLIST person holiday (yes/no) "no">
<!ELEMENT name ((first, surname)(surname, first))>
<!ELEMENT first (#PCDATA)>
<!ELEMENT surname (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT relationships EMPTY>
<!ATTLIST relationships superior IDREF #IMPLIED
inferior IDREFS #IMPLIED>

```

Fig. 1. An example of a DTD

```

<element xmlns="http://relaxng.org/ns/structure/1.0"
name="employees">
<oneOrMore>
<element name="person">
<element name="name">
<interleave>
<element name="first"> <text/> </element>
<element name="surname"> <text/> </element>
</interleave>
</element>
...
<attribute name="id">
<data type="ID"/>
</attribute>
...
</element>
</oneOrMore>
</element>

```

```

element employees {
element person {
element name {
element first { text } &
element surname { text }
},
...
attribute id { xs:ID },
} +
}

```

Fig. 2. An example of a RELAX NG schema

```

<schema xmlns="http://www.ascc.net/xml/schematron">
<pattern name="Conditions for list of employees">
<rule context="employees">
<assert test="person">No person found</assert>
</rule>
</pattern>
<pattern name="Conditions for one person">
<rule context="person">
<assert test="name">A person must have an
element name</assert>
<report test="name[2]">A person cannot have
multiple elements name</report>
<assert test="@id">A person must have an
attribute id</assert>
...
</rule>
</pattern>
...
</schema>

```

Fig. 3. An example of a Schematron schema

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="employees">
<xs:complexType>
<xs:sequence>
<xs:element ref="person" minOccurs="1"
maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>

```

```

<xs:element name="person">
<xs:complexType>
<xs:sequence>
<xs:element ref="name"/>
<xs:element name="email" type="xs:string"
minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="relationships" minOccurs="0"
maxOccurs="1"/>
</xs:sequence>
<xs:attribute name="id" type="xs:ID" use="required"/>
<xs:attribute name="note" type="xs:string"/>
<xs:attribute name="holiday" default="no">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:enumeration value="yes"/>
<xs:enumeration value="no"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>

```

```

<xs:element name="name">
<xs:complexType>
<xs:all>
<xs:element name="first" type="xs:string"/>
<xs:element name="surname" type="xs:string"/>
</xs:all>
</xs:complexType>
</xs:element>

```

```

<xs:element name="relationships">
<xs:complexType>
<xs:attribute name="superior" type="xs:IDREF"/>
<xs:attribute name="inferior" type="xs:IDREFS"/>
</xs:complexType>
</xs:element>
</xs:schema>

```

Fig. 4. An example of an XSD

lucid than the respective DTD. In addition, since the language involves a huge amount of constructs, which are mostly a “syntactic sugar” and, hence, have the same or almost the same expressive power, it is not easy for a user to learn all of them and decide which is better to use.

RELAX NG The authors of *RELAX NG* Murata (2002) tried to propose a language that involves key advantages of both DTD and XML Schema, but avoids their disadvantages. In particular, as we can see in Figure 2 the language has both XML and compact syntax which are equivalent and mutually transferable. Hence it exploits the advantages

of both DTD and XML Schema. Contrary to DTD it allows to specify the structure of a mixed-content element precisely (i.e. like XML Schema does); contrary to XML Schema it does not restrict the complexity of unordered sequences only to simple cases. Also, similarly to XML Schema, it supports a wide set of simple data types, both built-in and user-defined. On the other hand, precise occurrence ranges of elements or groups of elements are surprisingly not supported – RELAX NG supports the same operators as DTD (i.e. `optional`, `oneOrMore` and `zeroOrMore`). Last but not least, contrary to both DTD and XML Schema, the definition of a content model can combine both elements and attributes, hence the expressive power is much higher. XML Schema can express similar restrictions using assertions, but these were established in version 1.1 and currently are not much supported so far.

Schematron Contrary to the previous languages where we can find numerous similarities, *Schematron* Jelliffe (2001) uses a completely different strategy. However, on the other hand, it probably served as an inspiration for XML Schema assertions. As we can see in Figure 3, each Schematron schema is based on the idea of *patterns*. A pattern can be described as a set of *rules* an XML document must satisfy to be valid against a Schematron schema. A rule involves either element `assert` or `report` depending on the requirement of satisfaction or not satisfaction of the condition specified in attribute `test`. Since a Schematron schema can be evaluated by translation to XSLT Clark (1999) and usage of an XSLT parser, it supports the same subset of XPath Clark and DeRose (1999) as XSLT.

Even though the expressive power of Schematron is apparently higher than in case of the previous three XML schema languages (if we omit XML Schema assertions), Schematron has also several disadvantages. As we can see in Figure 3, the biggest problem is complexity of expressing simple rules such as “an element *e* has an attribute *a*”. Hence, Schematron should rather be considered as an extension of the previous languages that enables one to express complex *integrity constraints*. Both XML Schema and RELAX NG support Schematron subschemas.

3. Formal View of XML Schema Languages For the purpose of the following text we need a formal view of XML documents, XML schema languages and mutual validity. In general, we can divide the described schema languages into *grammar-based* (i.e. DTD, XML Schema, RELAX NG) and *pattern-based* (i.e. Schematron). The majority of current papers deal with basic and most common structural specification of XML data that can be expressed using the grammar-based languages. In other words, they do not deal with advanced integrity constraints that can be expressed using Schematron or XML Schema assertions. So, if not stated otherwise, we consider the same set.

The grammar-based XML schema languages we consider in the first parts of our text can be further classified according to their expressive power. We borrow and slightly modify for our purposes the definitions from Murata *et al.* (2005). Since the well-formedness of XML documents Bray *et al.* (2008) ensures that the correct usage of start and end tags forms a tree structure of XML documents, we usually speak about *tree grammars*. In addition, since most of the current approaches do not consider attributes, because their in-

ference can be considered as a special case of inference of elements having a text content, we omit them for simplicity too.

We represent an XML document as directed labeled tree $t = (V, E)$ called *XML tree* whose vertices from set V represent XML elements and attributes and edges from set E represent the hierarchical structure. We also consider a common formalization of an XML schema in a form of a *regular tree grammar (RTG)* $G = (N, T, S, P)$ having a set of non-terminals N , a set of terminals T , a set of start symbols S and a set of production rules P . Terminals of the grammar specify allowed elements (and attributes) in the XML document and the *production rules*, resp. *regular expressions (REs)* on their right hand sides, specify their allowed content. An *XML tree valid* against a given regular tree grammar is then an XML tree which can be constructed by the rewriting rules of the grammar. Another possibility to formalize XML schemas are classical *finite state automata (FSA)*. It is a well known fact that production rules of regular grammars can be expressed as finite state automata and vice versa.

In Murata *et al.* (2005), various classes of RTGs that correspond to particular XML schema languages were introduced. In particular, we can define so-called *local-tree grammars* that correspond to DTD and *single-type tree grammars* that correspond to XML Schema. Note that RELAX NG corresponds to general regular tree grammars.

Definition 3.1. *Let us have an RTG $G = (N, T, S, P)$. Two non-terminals $A, B \in N$ are competing with each other if there exist two production rules $A \rightarrow ar_1$ and $B \rightarrow ar_2$, where $a \in T$ and r_1, r_2 are REs over N .*

Definition 3.2. *A local tree grammar (LTG) is a RTG without competing non-terminals. A tree language is a local tree language if it is generated by a local tree grammar.*

Definition 3.3. *A single-type tree grammar (STTG) is a RTG such that*

- *for each production rule, non-terminals in its content model do not compete with each other, and*
- *start symbols do not compete with each other.*

A tree language is a single type tree language if it is generated by a single type tree grammar.

4. General Framework of Heuristic Inference Approaches The studied problem of XML schema inference can be described as follows: Being given an input set of XML trees $I = \{t_1, t_2, \dots, t_n\}$, we search for an XML schema, i.e. an RTG $G_I = (N_I, T_I, P_I, S_I)$, such that $\forall i \in [1, n] : t_i$ is valid against G_I . In particular, we are searching for G_I that is enough concise, precise and, at the same time, general. This requirement indicates, that the optimal result is hard to define and, in general, there may exist several solutions, i.e. a set of candidate RTGs $O = \{G_I^1, G_I^2, \dots, G_I^m\}$, such that $\forall i \in [1, n], \forall j \in [1, m] : t_i$ is valid against G_I^j , whereas we are looking for the optimal $G_I^{opt} \in O$.

The problem of finding G_I^{opt} can be viewed as a special kind of optimization problem called *combinatorial optimization problem (COP)* Barták (1998).

Most of the existing works use the same strategy consisting of the following phases to solve this problem:

- Phase I. Derivation of initial grammar
- Phase II. Clustering of production rules of initial grammar
- Phase III. Inference of REs
- Phase IV. Refactorization
- Phase V. Inference of simple data types
- Phase VI. Inference of integrity constraints
- Phase VII. Expressing the inferred items in the target XML schema language

The current methods differ in involving/omitting selected phases and, in particular, strategies they apply on them. In the following sections we describe in detail the current approaches and the improvements we have proposed so far.

4.1. Phase I. Derivation of Initial Grammar The first phase of the inference process is the same in all existing works: For each element node e in any of the XML trees in I and its child nodes e_1, e_2, \dots, e_k we construct a production \vec{p}_e of the form $e \rightarrow lab(e)(e_1 e_2 \dots e_k)$, where $lab(e)$ denotes the label of e . The production rules form so-called *initial grammar (IG)*.

Example 4.1. An example of an XML document (a) and respective IG (b) is depicted in Figure 5.

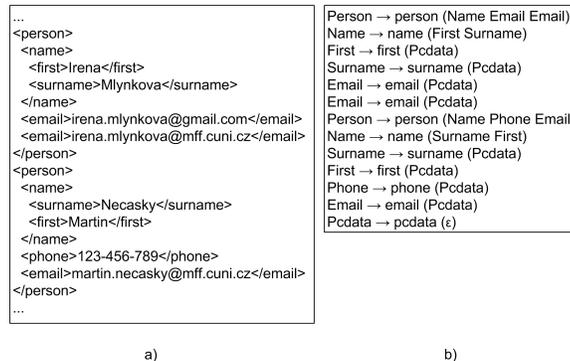


Fig. 5. An example of an XML document (a) and its IG (b)

Note that for the sake of clarity we start non-terminals of IG with capital letters and terminals of IG with small letters.

4.2. Phase II. Clustering of Production Rules of IG In the second phase the production rules of IG need to be clustered, since for each cluster a single RE is inferred in phase III (see Section 4.3). A typical strategy of the existing works is to cluster the production rules simply on the basis of the equivalence of left-hand sides.

Example 4.2. An example of clusters of IG in Figure 5 (omitting duplicities and the production rule for *pcdata*) is depicted in Figure 6.

Person → person (Name Email Email)
Person → person (Name Phone Email)
Name → name (First Surname)
Name → name (Surname First)
First → first (Pcdata)
Surname → surname (Pcdata)
Email → email (Pcdata)
Phone → phone (Pcdata)

Fig. 6. Production rules of IG clustered according to equivalence of left hand sides

Apparently, such beginning step leads to inference of an LTG, i.e. DTD, where all the elements are defined at the same level, and, hence, we are not able to specify elements with the same name but different structure. However, the described functionality is included in XML Schema and RELAX NG, i.e. STTGs and RTGs, and can have quite reasonable usage due to homonymy of element names.

Example 4.3. Let us have an XML schema of a library where each book, author and publisher has a name. In the former case it can be only a simple string, whereas in the latter two cases the name can consist of a couple of elements each having its own semantics – see Figure 7.

<pre><book> <name>Sherlock Holmes</name> </book></pre>	<pre><author> <name> <first>Arthur</first> <middle>Conan</middle> <last>Doyle</last> </name> </author></pre>	<pre><publisher> <name> Barnes and Noble <suffix>Inc.</suffix> </name> </publisher></pre>
--	--	---

Fig. 7. Elements with the same name but different structure

In paper Vošta *et al.* (2008) we have proposed two strategies that enable one to infer both STTGs and RTGs. They are based on two extensions of the clustering algorithm – on the basis of similarity of context of elements and similarity of content of elements. In paper Vyhnánovská and Mlýnková (2010) we have proposed another extension, that enables one to cluster also elements with different context (and names), but similar structure. This feature is supported in XML Schema, where various features (such as references or assigning of globally defined data types) enable one to create shared parts of the schema and thus describe the data more precisely. Both the extensions are described in the following subsections.

4.2.1. Similarity of Context The key feature of STTGs is based on the fact that elements in the same content model, i.e. context, do not compete with each other. First, we need a formal definition of a context.

Definition 4.1. A context $cont_v$ of an element node $v \in V$ in a document tree $t = (V, E)$ is a concatenation of $lab(v_0)lab(v_1)...lab(v_n)$, where v_0 is the root node of the tree, $v_n = v$, and $\forall i \in [1, n] : \langle v_{i-1}, v_i \rangle \in E$.

For the purpose of evaluation of similarity of two contexts $cont_v$ and $cont_u$, we exploit and modify the classical Levenshtein algorithm Levenshtein (1966) that determines the edit distance of two strings s_x, s_y using inserting, deleting or replacing a single character. Each operation is assigned with a constant cost and the edit distance (i.e. similarity) is given by the amount of operations that need to be done to transform s_x to s_y . In our case instead of single characters our operations work with whole element names.

Hence, in the following text we can assume that we have a function $sim_{context} : V \times V \rightarrow [0, 1]$ which expresses the similarity of contexts of two element nodes, where 0 denotes strong dissimilarity and 1 equivalence.

4.2.2. Similarity of Content The key feature of RTGs is that they allow for any kind of competing non-terminals. In other words, we can cluster the elements not only according to their context, but also content. As an XML element e can be viewed as a subtree t_e (in the following text denoted as an *element tree*) of corresponding document tree t , we use a modified idea of *tree edit distance*, where the similarity of trees t_e and t_f is expressed using the minimum number of edit operations necessary to transform t_e into t_f (or vice versa).

Currently, there exist several approaches to tree edit distance (e.g. Touzet (2005); Marian (2002); Torsello and Hancock (2003); Touzet (2003)). The key aspect is obviously the set of allowed edit operations which need to be suitable for a particular application.

Example 4.4. Consider two simple operations – adding and removal of a leaf node (and respective edge). As depicted in Figure 8, such similarity is not suitable, e.g., for recursive elements. The example depicts two element trees of element a having subelement i having subelement j having subelement k which contains either subelement z or again i . With the two simple edit operations the edit distance would be 4, but, since the elements have the same XML schema, we would expect the optimal distance of 1 reflecting the usage of one additional recursive level.

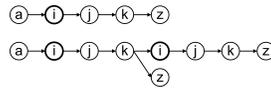


Fig. 8. Tree edit distance of recursive elements

For our purpose we exploit a similarity measure defined in Nierman and Jagadish (2002) which specifies more complex XML-aware tree edit operations on whole subtrees, each having its constant cost, as follows:

- *Insert* – a single node n is inserted to the position given by parent node p and ordinal number expressing its position among subelements of p

- *Delete* – a leaf node n is deleted
- *Relabel* – a node n is relabeled
- *InsertTree* – a whole subtree t is inserted to the position given by parent node p and ordinal number expressing position of its root node among subelements of p
- *DeleteTree* – a whole subtree t rooted at node n is deleted

As it is obvious, for given trees t_e and t_f there are usually several possible transformation sequences for transforming t_e into t_f . A natural approach is to evaluate all the possibilities and to choose the one with the lowest cost. But such approach can be quite inefficient. Thus authors of Nierman and Jagadish (2002) propose so-called *allowable sequences* of edit operations, which significantly reduce the set of possibilities and, at the same time, speed up their cost evaluation.

Definition 4.2. *A sequence of edit operations is allowable if it satisfies the following two conditions:*

1. *A tree t may be inserted only if t already occurs in the source tree t_e . A tree t may be deleted only if it occurs in the destination tree t_f .*
2. *A tree that has been inserted via the *InsertTree* operation may not subsequently have additional nodes inserted. A tree that has been deleted via the *DeleteTree* operation may not previously have had children nodes deleted.*

The first restriction forbids undesirable operations like, e.g., deleting whole t_e and inserting whole t_f etc., whereas the second one enables one to compute the costs of the operations efficiently. The evaluating algorithm is based on the idea of determining the minimum cost of each required insert of every subtree of t_e and delete of every subtree of t_f using a simple bottom-up procedure.

In the following text we assume that we have a function $sim_{content} : V \times V \rightarrow [0, 1]$ which expresses the similarity of element trees t_e and t_f rooted at element nodes $e, f \in V$, i.e. content of elements.

4.2.3. Clustering Algorithm In the resulting clustering algorithm we utilize a modification of classical *mutual neighborhood clustering (MNC) approach* Jain and Dubes (1988). We start with initial clusters c_1, c_2, \dots, c_k of elements given by the equivalence of their context, i.e. elements e and f belong to cluster c_i if $sim_{context}(e, f) = 1$. In other words, the initial clustering is based on a natural assumption that elements having the same context are likely to have the same schema definition. The initial clusters are then merged on the basis of element structure using the tree edit distance $sim_{content}$. Firstly, $\forall i \in [1, k]$ we determine a representative element r_i of cluster c_i . Then, for each pair of $\langle r_i, r_j \rangle$ such that $i, j \in [1, k]; i \neq j$ we determine tree edit distance $dist_{content}(r_i, r_j)$ of the respective trees. The MNC algorithm is parameterized by three parameters – minimum distance $dist_{MIN}$, maximum distance $dist_{MAX}$, and factor ϕ – and exploits the definition of *mutual neighborhood*:

Definition 4.3. *Let e and f be two elements, where e is i -th closest neighbor of f and f is j -th closest neighbor of e . Then mutual neighborhood of e and f is defined as $MN(e, f) = i + j$.*

Consequently, the MNC algorithm places two elements r_i and r_j into the same cluster if:

$$\begin{aligned} & dist_{content}(r_i, r_j) \leq dist_{MIN} \vee \\ & (dist_{content}(r_i, r_j) \leq dist_{MAX} \wedge MN(r_i, r_j) \leq \phi) \end{aligned} \quad (1)$$

resulting in a set of clusters c_1, c_2, \dots, c_l (where $l \leq k$) of elements grouped on the basis of their context and content.

4.3. Phase III. Inference of REs Having the set of clusters c_1, c_2, \dots, c_l of production rules, the key emphasis is in the existing works put on inference of REs. A common approach is so-called *merging state algorithm* which consists of two steps:

1. $\forall i \in [1, l]$ a *prefix tree automaton (PTA)* A_{c_i} is built from the production rules of cluster c_i .
2. Each A_{c_i} is generalized via merging of its states.

Example 4.5. An example of a cluster c_i of IG and the respective PTA A_{c_i} is depicted in Figure 9.

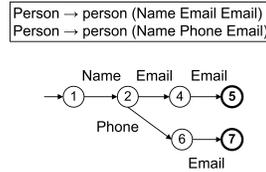


Fig. 9. An example of an IG and a PTA

In Mlýnková (2009) we have focussed on further extension of this step based on a typical real-world situation when a user creates an XML schema of XML documents but then modifies and updates only the data (identified in Mlýnková *et al.* (2006)). In other words, we are provided not only with the input documents $I = \{t_1, t_2, \dots, t_n\}$, but also an *obsolete XML schema*, i.e. in general an RTG $G_{obs} = (N_{obs}, T_{obs}, P_{obs}, S_{obs})$, such that there exist parts of the input documents that are valid against it, i.e. the similarity of G_{obs} and G_I^{opt} is high. Such input can be exploited for speeding up the inference process. We only need to modify the process of creating the initial automata to be optimized and generalized. In particular, we cluster and merge the production rules of IG extracted from I together with production rules from P_{obs} . Hence, we do not start with PTAs, but general FSAs.

Example 4.6. An example of merging a production rule of IG with a production rule from an obsolete schema G_{obs} is depicted in Figure 10.

Regardless the input automaton, the rules and strategies for merging the states of a PTA used in the current approaches differ, but they have a common aim to output a

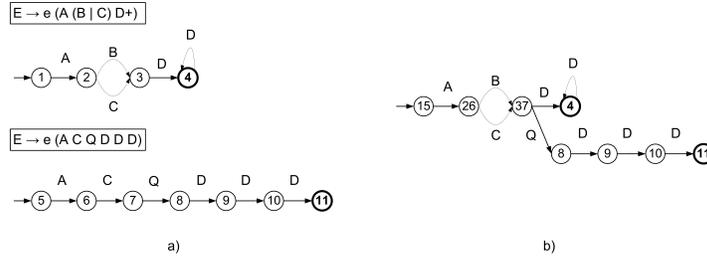


Fig. 10. Exploitation of an obsolete schema. a) a production rule of an IG and a production rule of an obsolete schema, b) result of their merging.

concise and precise XML schema. As we have mentioned at the beginning of this section, since the amount of possible output automata, i.e. schemas, is theoretically infinite, the approaches search only a subspace of possible solutions using a kind of greedy-search, terminating condition etc.

4.3.1. Naive Solutions The first and simplest approaches implemented, e.g., in systems *GB-engine* Shafer (1995) or *DTD-miner* Moh *et al.* (2000), use a simple set of heuristic rules such as those depicted in Figure 11 and the search strategy continues until there exists a rule that can be applied.

$aaaa$	\Rightarrow	a^+
$(ab)^* a?b?$	\Rightarrow	$(a?b?)^*$
$ab ab^*$	\Rightarrow	ab^*
$a?, b?, c?$	\Rightarrow	$a b c$
a, b, c, d, a, d, b, c	\Rightarrow	$(a b c d)^+$

Fig. 11. Simple heuristic rules for merging states of automaton

4.3.2. Evaluation of Schema Quality A strategy similar to *DTD-miner* is used also in system *XTRACT* Garofalakis *et al.* (2000). However, the rules are not applied using a greedy search, but a set of possible solutions is produced and the system selects the optimal one, i.e. it is able to evaluate quality of a schema generalization.

For the purpose of schema evaluation the authors (as well as the subsequent approaches) exploit so-called *minimum description length (MDL) principle* Grunwald (2005). It expresses the quality of an XML schema candidate using two aspects – conciseness and preciseness. *Conciseness* of an XML schema is expressed using the number of bits required to describe the schema itself (the smaller, the better). *Preciseness* of an XML schema is expressed using the number of bits required for description of the input XML trees in I using the schema. In other words, on the one hand, a good schema should be enough general which is related to the low number of states of the schema automaton, but, on the other hand, it should preserve details which means that it enables one to express document instances using short codes, since most of the information is carried by the schema itself.

Formally, we can express the quality of an RTG $G_{c_i} = (N_{c_i}, T_{c_i}, S_{c_i}, P_{c_i})$ inferred from a set of XML trees in a cluster c_i as the sum of the size (in bits) of P_{c_i} and the size of codes of instances in c_i . Let D be the set of allowed operators and E the set of distinct element names in c_i . Then we can view the right-hand side of each $p \in P_{c_i}$ as a word over $D \cup E$ and the size of its code can be expressed as:

$$|p| \times \lceil \log_2(\text{card}(D) + \text{card}(E)) \rceil \quad (2)$$

where $|p|$ denotes length of word p . The size of code of a single element tree $e \in c_i$ is defined as the size of code of a sequence of production rules $P_{\langle e \rangle} = \langle \vec{p}_1, \vec{p}_2, \dots, \vec{p}_k \rangle$ necessary to convert an initial nonterminal $s \in S_{c_i}$ to e using production rules from P_{c_i} . Since we can represent the sequence $P_{\langle e \rangle}$ as a sequence of ordinal numbers of the production rules in P_{c_i} , the size of the code of e can be expressed as:

$$\text{card}(P_{\langle e \rangle}) \times \lceil \log_2(\text{card}(P_{c_i})) \rceil \quad (3)$$

4.3.3. Advanced Merging Rules Apart from simple heuristic merging rules, there exist also approaches that base their strategy on various theoretical results.

The *k, h-context method* Ahonen (1996) specifies an identifiable subclass of regular languages which assumes that the context of elements is limited. So merging states of an automaton $A = (Q, \Sigma, \delta, s, F)$ is based on an assumption that two states $x, y \in Q$ are identical (and can be merged) if there exist two identical paths of length k terminating in x and y . In addition, also $h \leq k$ preceding states in these paths are then identical.

On the other hand, the *s, k-string method* Wong and Sankey (2003) is based on Nerod's equivalency of states of automaton A assuming that two states $x, y \in Q$ are equivalent if sets of all paths leading from x and y to any $f \in F$ are equivalent. But since such condition is hard to check, we can restrain to k -strings, i.e. only paths of length of k or terminating in a terminal state $f \in F$. The respective equivalency of states then depends on equivalency of sets of outgoing k -strings. In addition, for easier processing we can consider only s most probable paths, i.e. we can ignore singular special cases.

Example 4.7. *Two examples of the described merging are depicted in Figure 12. In the former case (a), we can merge states 4 and 5, because they have the same prefixes of length 1 (i.e. edge Address). In the latter case (b), we can merge states 3, 5 and 4, 6 because they have the same suffixes of length 1.*

4.3.4. Advanced Search Strategies As we have mentioned, we can view the problem of generalization of an automaton as a kind of optimization problem where we search for an optimal solution in a theoretically infinite space. The basic strategy is to use a kind of a greedy search. However, with the general view of the problem as a COP, we can use also more advanced meta-heuristics.

In Wong and Sankey (2003) the authors utilize a classical approach called *Ant Colony Optimization (ACO)* Dorigo *et al.* (2006). The ACO heuristic is based on observations of nature, in particular the way ants exchange information they have learnt. A set of artificial "ants" Δ search the space of solutions Θ trying to find the optimal solution

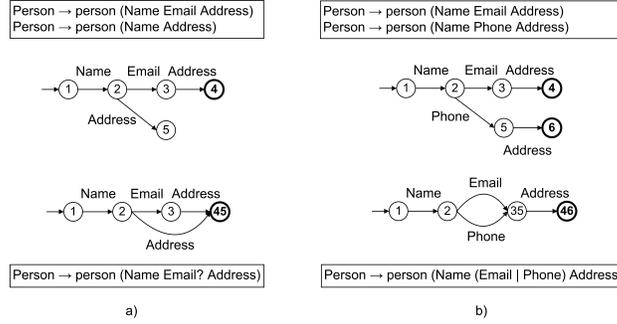


Fig. 12. Merging states of an automaton. a) k, h -context and b) s, k -string.

$s_{opt} \in \Theta$ such that $\sigma(s_{opt}) \leq \sigma(s); \forall s \in \Theta$. In i -th iteration each ant $a \in \Delta$ searches a subspace of Θ for a local suboptimum until it “dies” after performing a predefined amount of steps N_{ant} . While searching, an ant a spreads a certain amount of “pheromone”, i.e. a positive feedback which denotes how good solution it has found so far. This information is exploited by ants from the following iterations to choose better search steps. The key aspect of the algorithm is one step of an ant. Each step consists of generating of a set of possible movements, their evaluation using σ , and execution of one of the candidate steps. The executed step is selected randomly on the basis of probability given by σ . The algorithm terminates either after a specified number of iterations N_{iter} or if $s'_{opt} \in \Theta$ is reached such that $f(s'_{opt}) \leq T_{max}$, where T_{max} is a required threshold. Note that the randomness is the key aspect of the metaheuristic, since it enables one to search larger space of solutions than greedy strategies do and thus possibly find a better suboptimum.

In Vošta *et al.* (2008) we have exploited also a temporary negative feedback which enables one to search a larger subspace of Θ . The idea is relatively simple – whenever an ant performs a single step, it spreads a reasonable negative feedback. The difference is that the positive feedback is assigned after i -th iteration is completed, i.e. all ants die, to influence the behavior of ants from $(i + 1)$ -st iteration. The negative feedback is assigned immediately after a step is performed, i.e. it influences behavior of ants in i -th iteration and at the end of the iteration it is zeroed.

In our case of schema inference we start with the PTA (or its modification) from phase II (see Section 4.2). A step of an ant means application of a selected generalization rule, either a naive one (see Section 4.3.1) or any of the advanced methods such as, e.g., k, h -context or s, k -string (see Section 4.3.3). The objective function σ can be defined, e.g., using the MDL principle (see Section 4.3.2).

4.3.5. XML Schema Unordered Sequences In all the previous approaches the authors focussed on constructs that can be expressed in DTD. In Vošta *et al.* (2008) we extended the current approaches with the ability to infer also unordered sequences of elements, i.e. XML Schema `all` construct (or `&` operator for simplicity). (Even though, such construct can be expressed in DTD using a list all possible permutations, such approach is not used in practise for apparent disadvantages.) Our extension is based on the

observation that it can be considered as a special kind of rule for merging states of an automaton. It enables one to replace a set of ordered sequences of elements with a single unordered sequence represented by the $\&$ operator.

In general the $\&$ operator can express the unordered sequence of REs of any complexity such as, e.g., $(e_1|e_2) * \&e_3? \&(e_4, e_5, e_6)$. But, the W3C recommendation of XML Schema language does not allow to specify so-called *nondeterministic data model* Mani (2001), i.e. a data model which cannot be matched without looking ahead. A simple example can be a RE $(e_1, e_2)|(e_1, e_3)$, where while reading the element e_1 we are not able to decide which of the alternatives to choose unless we read the following element. Hence, also the allowed complexity of unordered sequences is restricted. The former and currently recommended version 1.0 of XML Schema specification Thompson *et al.* (2004); Biron and Malhotra (2004) allows to specify an unordered sequence of elements, each with the allowed occurrence of $[0, 1]$, whereas the allowed occurrence of the unordered sequence itself is of $[0, 1]$ too. XML Schema 1.1 Gao *et al.* (2009); Peterson *et al.* (2009), currently in the status of a working draft, is similar but the allowed number of occurrence of items of the sequence is $[0, \infty]$. In our approach we focus on the more general possibility, since it will probably soon become a new recommendation.

For the purpose of identification of subgraphs representing the allowed type of unordered sequences, we first define so-called *common ancestors* and *common descendants*.

Definition 4.4. Let $G = (V, E)$ be a directed graph. A common descendant of a node $v \in V$ is a descendant $d \in V$ of v such that all paths traversing v traverse also d .

Definition 4.5. Let $G = (V, E)$ be a directed graph. A common ancestor of a node $v \in V$ is an ancestor $a \in V$ of v such that all paths traversing v traverse also a .

Definition 4.6. Let $G = (V, E)$ be a directed graph. A common ancestor of a node $v \in V$ with regard to a node $u \in V$ is an ancestor $a \in V$ of v such that a is a common ancestor of each direct ancestor of v occurring on path from u to v .

Example 4.8. Considering the example in Figure 13 (temporarily without labels of edges) we can see that the common descendants of node 3 are nodes 6 and 7, whereas node 1 has no common descendants, since paths traversing node 1 terminate in nodes 7 and 9. Similarly, the common ancestor of node 6 is node 1. Note that in the former case there can exist paths which traverse d but not v (see node 3 and its common descendant 6), whereas in the latter case there can exist paths which traverse a but not v . The common ancestors of node 6 with regard to node 2 are nodes 2 and 3.

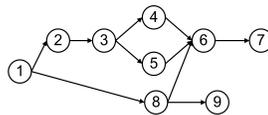


Fig. 13. An example of common ancestors and descendants

We denote the node v from Definition 4.4 or the node a from Definitions 4.5 and 4.6 as *input nodes* and their counterparts as *output nodes*. The set of nodes occurring on paths starting in an input node and terminating in an output node are called a *block*. Using the definitions we can now identify subgraphs which are considered as *first-level candidates* for unordered sequences.

Definition 4.7. Node n_{in} is an input node of block representing a first-level candidate if:

1. its out-degree is higher than 1,
2. the set of its common descendants is not empty, and
3. at least one of its common descendants, denoted as n_{out} , whose set of common ancestors with regard to n_{in} contains n_{in} .

The three conditions ensure that there are at least two paths leading from n_{in} representing at least two alternatives and that the block is *complete*, meaning that there are no paths entering or leaving the block otherwise than using n_{in} or n_{out} .

Example 4.9. Considering Figure 13, the only first-level candidate is subgraph consisting of nodes 3, 4, 5, and 6.

Having a first-level candidate we need to check it for fulfilling conditions of an unordered sequence and, hence, being a so-called *second-level candidate*. We again exploit the idea of similarity of graphs expressed using edit distance. In particular, we exploit the fact that for each $n \in \mathbb{N}$ we know the structure of the automaton \mathcal{P}_n which accepts each permutation of n items having all the states fully merged.

Example 4.10. An example of automaton \mathcal{P}_3 is depicted in Figure 14 for three items A , B , C .

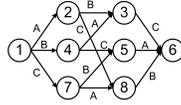


Fig. 14. Automaton \mathcal{P}_3 – permutation of three items

The idea is to compare the similarity of the first-level candidates with \mathcal{P}_n automata. We only need to modify the first-level candidate so that the similarity measure can cope with optional and repeatable elements in the unordered sequences and the fact that the input elements on whose basis the automaton was built do not need to contain all possible permutations. We can also observe, that the candidate graph must be always a subgraph of \mathcal{P}_n , otherwise we can skip its processing. Hence, the problem of edit distance is highly simplified and we use the following types of edit operations:

- adding an edge between two existing nodes, and
- splitting an existing edge into two edges, i.e. adding a new node and an edge.

The first operation is obvious and corresponds to the operation of adding paths that represent permutations which were not present in the source data. In the latter case the operation corresponds to adding an item of a permutation which was not present in the source data.

4.3.6. XML Schema “Syntactic Sugar” As we have mentioned, XML Schema involves plenty of “syntactic sugar”, i.e. constructs being *equivalent*. For instance, the unordered sequences described in the previous section represent one of the cases.

Definition 4.8. Let s_x and s_y be two XML schema fragments. Let $\eta(s) = \{d \text{ such that } d \text{ is an element tree valid against } s\}$. Then s_x and s_y are equivalent, denoted as $s_x \sim s_y$, if $\eta(s_x) = \eta(s_y)$.

Example 4.11. As depicted in Figure 15 (a), there is no difference if a simple type is defined locally or globally. Example (b) depicts the equivalence between an unordered sequence of a set of elements and a choice of their possible ordered permutations.

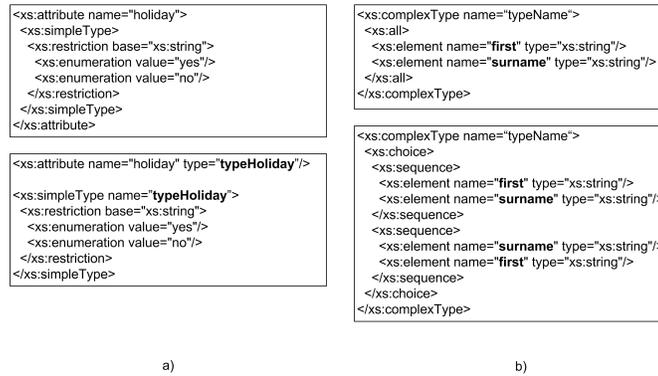


Fig. 15. “Syntactic sugar” of XML Schema. a) globally and locally defined simple types, b) unordered sequences.

Consequently, having a set ξ of all XSD constructs, we can specify the quotient set ξ / \sim of ξ by \sim and respective equivalence classes Mlýnková and Nečaský (2009) – see Table 1. Each of the classes of \sim equivalence can be then represented using a selected *canonical representative* as listed in Table 1 as well. Note that each of the constructs not mentioned in the table forms a single class C_1, C_2, \dots, C_n .

With regard to the classes, we can generalize the previous approach for inference of unordered sequences, since from a general point of view we can find multiple ways how to express an XSD. For our purposes, we first slightly modify its definition Vyhnanovská and Mlýnková (2010).

Definition 4.9. An extended finite state automaton (*exFSA*) is a 6-tuple $(Q, Q_{ex}, \Sigma, \delta, s, F)$, where:

Table 1. XSD equivalence classes of ξ/\sim

Class	Constructs	Canonical representative
C_{ST}	globally defined simple type, locally defined simple type	locally defined simple type
C_{CT}	globally defined complex type, locally defined complex type	locally defined complex type
C_{El}	referenced element, locally defined element	locally defined element
C_{At}	referenced attribute, locally defined attribute, attribute referenced via an attribute group	locally defined attribute
C_{ElGr}	content model referenced via an element group, locally defined content model	locally defined content model
C_{Seq}	unordered sequence of elements e_1, e_2, \dots, e_l , choice of all possible ordered sequences of e_1, e_2, \dots, e_l	choice of all possible ordered sequences of e_1, e_2, \dots, e_l
C_{CTDer}	derived complex type, newly defined complex type	newly defined complex type
C_{SubGr}	elements in a substitution group γ , choice of elements in γ	choice of elements in γ
C_{Sub}	data types $\tau_1, \tau_2, \dots, \tau_k$ derived from type τ , choice of content models defined in $\tau_1, \tau_2, \dots, \tau_k, \tau$	choice of content models defined in $\tau_1, \tau_2, \dots, \tau_k, \tau$

- Q is a set of basic states,
- Q_{ex} is a set of extended states,
- Σ is a set of input symbols,
- $\delta : Q \cup Q_{ex} \times \Sigma^* \rightarrow Q \cup Q_{ex}$ is the transition function,
- $s \in Q \cup Q_{ex}$ is the initial state, and
- $F \subseteq Q \cup Q_{ex}$ is the set of final states.

An exFSA behaves similarly to a deterministic FSA. The only difference is that it contains in addition a set of *extended states*. An extended state s_x represents a subautomaton A_x which accepts a part of the input XML tree. When the s_x state is reached, the A_x automaton continues in processing of input, it *consumes* as much symbols as possible and then returns processing to the original automaton. Every extended state has a *helper state*, i.e. a basic state with only one transition point to it – the λ -transition from the extended state. When the processing is returned from the subautomaton, the automaton moves into the helper state according to the λ -transition.

An exFSA has the same expressive power as a FSA and it can be transformed to the FSA with λ -edges. An extended state has only one transition – λ -transition – which is redirected to the initial state of the subautomaton. All final states of the subautomaton have one additional λ -transition to the destination state of the original λ -transition.

Example 4.12. An example of exFSA and its transformation to the respective FSA is depicted in Figure 16.

An exFSA enables us to *outline* and merge equivalent schema fragments, i.e. to express *sharing* of selected parts of an XSD. However, we cannot merge and outline schema

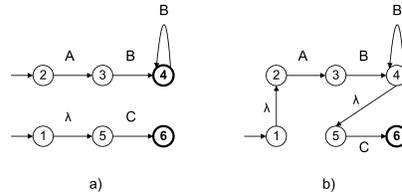


Fig. 16. exFSA transformed to the FSA. a) up – a subautomaton, down – exFSA with an extended state 1 and helper state 5, b) transformed FSA with λ -edges.

fragments that have nothing in common but a subset of subelements.

Example 4.13. Let us have an element *person* with attribute *id* and subelements *name*, *address* and *phone* and an element *book* with attribute *id* and subelements *name*, *address* and *authors*. The approach would lead to creating three complex types – *typeCommon* involving the common items and two derived types *typePerson* and *typeBook*, each adding the respective subelement. Since the two elements have nothing in common, such schema would obviously be semantically wrong.

The solution to the problem we proposed Mlýnková and Nečaský (2009) is taking into account also the semantics of the data. In particular, we use a kind of a thesaurus that enables one to discover semantic relations. Hence, we discover that the semantic similarity of words *person* and *book* is not high, but similarity of, e.g., *person*, *author* and *editor* is sufficient. And it also brings another advantage, since it indicates the way we should express the sharing (as discussed in detail in Section 4.7).

Example 4.14. Let us have two candidates for outlining – *employee* and *manager*. The thesaurus not only determines that they are related, but also that *employee* is a broader term to *manager*. Hence, instead of creating *typeCommon* and derived types *typeEmployee* and *typeManager*, we create *typeEmployee* and a derived type *typeManager*.

To ensure the indicated functionality, we extend the process of inference of RE with an outlining rule. The algorithm directly searches for equivalent schema fragments and analyzes their semantic relevance. If it is identified as sufficient, the respective subautomaton is outlined.

Example 4.15. An example of outlining of an automaton is depicted in Figure 17 (a) and (b). In this case we exploit the fact that elements *person* and *manager* have common items that can be outlined into a globally defined schema fragment represented using a subautomaton. Since “*person*” and “*manager*” are semantically related, the sets are enough large and the items are not too general, the result again bears more information and it is more realistic.

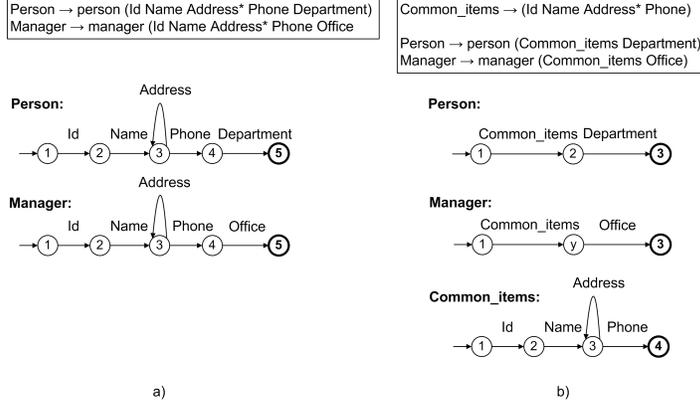


Fig. 17. Splitting and outlining an automaton. a) productions with a common fragment, b) outlined common fragment.

Evaluation of Quality of a Schema With the introduced extension, we need to modify the strategy for evaluation of quality of the given XML schema. Otherwise, our more precise XML schema would never be inferred, because since our primary aim is conciseness, their quality with regard to MDL (see Section 4.3.2) is lower.

Let us have an RTG $G_{c_i} = (N_{c_i}, T_{c_i}, S_{c_i}, P_{c_i})$ inferred from a set of XML trees in cluster c_i . Let us denote $P''_{c_i} \subseteq P_{c_i}$ the set of outlined production rules and $P'_{c_i} \subseteq P_{c_i}$ the set of original production rules from which a fragment was outlined. In general, if we outline a reasonably big schema fragment from at least two production rules, we obviously decrease the sum of sizes of production rules in P_{c_i} . But, on the other hand, if there exists an element tree $e \in c_i$ such that $P_{\langle e \rangle}$ involves a production rule $\vec{p} \in P_{c_i}$ that was split into multiple production rules, the length of $P_{\langle e \rangle}$ increases. To solve this problem, we modify the evaluation as follows:

- Each $\vec{p} \in P_{c_i}$ is provided with a weight $\alpha(p) \in [0, 1]$ which influences (multiplies) the size of code of \vec{p} expressing its appropriateness with regard to the previously described outlining heuristics.
- The production rules from P''_{c_i} are not involved in counting the size of $P_{\langle e \rangle}$ for $\forall e \in c_i$.

The strategy of assigning α is as follows: At the beginning of the search algorithm (i.e. when $P_{c_i} = P'_{c_i}$) for $\forall \vec{p} \in P'_{c_i} : \alpha(\vec{p}) = 1$ representing the fact that each contributes to the total cost 100% as in the original metric. If two production rules $\vec{p}, \vec{q} \in P_{c_i}$ are split into production rules $\vec{o} \in P''_{c_i}$ and $\vec{p}', \vec{q}' \in P_{c_i}$, such that $type(\vec{o}) \in model(\vec{p})$ and $type(\vec{o}) \in model(\vec{q}')$, each of them is assigned α as follows:

$$\begin{aligned} \alpha(\vec{o}) &= new \\ \alpha(\vec{p}') &= \alpha(\vec{p}); \vec{p}' \in P'_{c_i} \end{aligned} \quad (4)$$

$$= new; \vec{p} \in P''_{c_i} \quad (5)$$

$$\begin{aligned} \alpha(\vec{q}') &= \alpha(\vec{q}); \vec{q} \in P'_{c_i} \\ &= new; \vec{q} \in P''_{c_i} \end{aligned} \quad (6)$$

The new values are counted as follows: Let $context(\vec{x}) = \{\vec{q} \in P'_{c_i} \text{ such that } type(\vec{x}) \in model(\vec{q}) \vee (\exists \vec{p} \in P''_{c_i} \text{ such that } type(\vec{x}) \in model(\vec{p}) \wedge type(\vec{q}) \in context(\vec{p}))\}$ and let $model'(\vec{x})$ be $model(\vec{x})$, where $\forall \vec{p} \in P''_{c_i}$ such that $type(\vec{p}) \in model(\vec{x})$ is recursively replaced with $model(\vec{p})$. The weight $\alpha(\vec{x})$ is evaluated as follows:

$$\alpha(\vec{x}) = 1 - \left(\alpha_1 \times sim_{i=1}^{context(\vec{x})}(\vec{q}_i \in context(\vec{x})) + \alpha_2 \times \frac{|context(\vec{x})|}{|P'_{c_i}|} + \alpha_3 \times \frac{|model'(\vec{x})| - |stolist(model'(\vec{x}))|}{avg_{i=1}^k(|model'(\vec{q}_i)|)} \right) \quad (7)$$

where $\sum_{i=1}^3 \alpha_i = 1$ and $\forall i : \alpha_i \in [0, 1]$, $sim()$ evaluates semantic similarity of the given words and $stolist()$ returns the set of terminals of the given content model that occur in the stoplist. In other words, the weight of the outlined production rule is expressed as a combination of similarity of elements it influences (the higher, the better), the amount of elements it influences (the more, the better) and its size except for words in the stoplist (the bigger, the better).

Example 4.16. An example of counting the respective parameters for evaluation of α is depicted in Table 2. In step 1 we start with three production rules without parameters, since their weight α remains 1. In step 2 we outline production \vec{P} . It occurs in context of two original productions and its length is of 6. In step 3 we outline production \vec{Q} which occurs in context of \vec{C} , but also \vec{P} , resulting in overall context \vec{A} , \vec{B} and \vec{C} . The length of \vec{Q} is of 3. Note that the length of $model'(\vec{P})$ does not change though $model(\vec{P})$ does.

Table 2. An example of evaluation of weight α

Step	\vec{x}	$context(\vec{x})$	$ model'(\vec{x}) $
1	A \rightarrow a (U W (B C)) B \rightarrow b (X Y W (B C)) C \rightarrow c (D (B C))	- - -	- - -
2	A' \rightarrow a (U P) B' \rightarrow b (X Y P) C \rightarrow c (D (B C)) P \rightarrow (W (B C))	- - - \vec{A}, \vec{B}	- - - 6
3	A' \rightarrow a (U P) B' \rightarrow b (X Y P) C' \rightarrow c (D Q) P \rightarrow (W Q) Q \rightarrow (B C)	- - - \vec{A}, \vec{B} $\vec{A}, \vec{B}, \vec{C}$	- - - 6 3

4.3.7. Statistical Analysis of Input Documents If we want to go in the inference process even further and be more precise, we can exploit the given XML documents more deeply Mlýnková and Nečaský (2009). Our motivation results from ideas used in adaptive schema-driven XML-to-relational mapping strategies (e.g. Du *et al.* (2004)). They apply various XML-to-XML transformations (such as, e.g. $a^+ = a, a^*$) on the input XML schema, evaluate them and select the optimal one. For this purpose, we can exploit almost any equation known for regular expressions. However, since the amount of options is again large and we would get to the same problem as described above, we need to restrict ourselves to cases that can be assessed as relevant. But, since we do not have an etalon in a set of XML queries like the adaptive mapping methods do, we exploit etalons relevant to schema inference.

Example 4.17. *For instance we can find out that in 95% of cases element `person` has two subelements `phone` at maximum and only in 5% of cases there occur elements `person` having more than five subelements `phone`. In the existing works the schema would be generalized to `phone+` or `phone*` although for most of the input XML documents it is too general. Consequently, if we exploit the above described equation we could preserve the first two occurrences of element `phone` and provide more realistic schema, i.e. `phone phone phone*`, bearing more precise information.*

For the purpose of the indicated improvements we further extend the rules for merging states of an automaton. In particular, we add new merging rules that exploit statistics of the given XML data, i.e. while merging we take into account also additional information that influence the process.

Example 4.18. *An example of exploitation of data statistics is depicted in Figure 18. The numbers above the particular elements depict the amount of data instances that induce this production rule. If we do not consider the statistics, we would infer the schema (a). But, with regard to the data, it would be too general, since most of the `person` elements have right three subelements `phone`, whereas only in few cases there are persons with more phone numbers. Consequently, from the point of view of preciseness of information on the data, the schema (b) is much realistic and bears more precise information.*

Evaluation of Quality of a Schema Also in this case we need to modify the evaluations process, however we encounter an opposite problem. The length of the preferred production rule is always higher than the length of the most concise one, because (regarding the data statistics) we do not include all the repeating fragments into the repetition. We again provide the production rules with a weight, β , expressing their appropriateness with regard to splitting repetitions. The weight is assigned as follows: At the beginning of the search algorithm, each edge of the PTA is assigned a number of instances it is induced by. At the same time, each of the production rules is assigned weight β of 1 representing the fact that each contributes to the total cost 100% as in the original metric. If the selected merging rule does not induce merging repetitions, β remains the same and we only sum up the numbers of instances of the merged edges (as depicted in Figure 18). If a merging rule that induces repetitions is applied on a production rule \vec{x} , i.e.

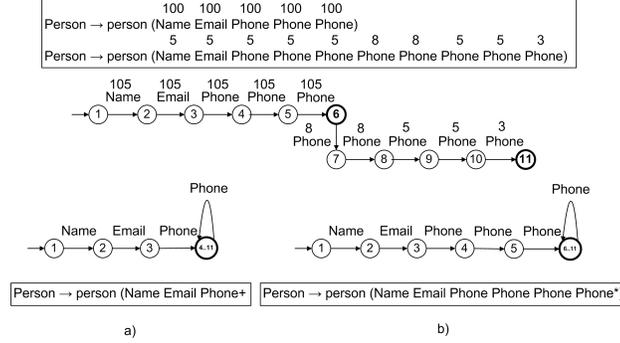


Fig. 18. Splitting repetitions. a) application of merging rule $a, a, a, \dots, a \Rightarrow a^+$, b) application of splitting repetitions rule $a, a, a, \dots, a \Rightarrow a, a, a, a^*$

there is a sufficient amount of a repeating pattern r (a sequence $r_{(1)}r_{(2)}\dots r_{(l)}$), the weight $\beta(\vec{x}')$ of resulting production rule \vec{x}' is counted as follows: Let rep_{min} be the minimum amount of occurrences a pattern must fulfill to be merged into a repetition, let i, j , such that $1 \leq i \leq j \leq l; j - i + 1 \geq rep_{min}$, determine the borders of the selected subsequence of repeating patterns to be merged and let $ind(r_{(i)})$ denote the number of inducing instances of $r_{(i)}$. Then:

$$\beta(\vec{x}') = 1 - \frac{score(\vec{x}')}{score(\vec{x})} \quad (8)$$

$$score(\vec{x}) = \sum_{k=1}^l ind(r_{(k)}) \quad (9)$$

$$score(\vec{x}') = \sum_{k=1}^{i-1} ind(r_{(k)}) + rep_{min} \times \max_{k=i}^j (ind(r_{(k)})) + \sum_{k=j+1}^l ind(r_{(k)}) \quad (10)$$

Example 4.19. Consider the example in Figure 18. If we do apply merging rule (a), assuming that $rep_{min} = 3$, then $score(\vec{x}) = 105 + 3 \times 105 = 420$. If apply merging rule (b), then $score(\vec{x}) = 105 + 105 + 105 + 3 \times 8 = 339$, i.e. it is a better candidate.

4.4. Phase IV. Refactorization A natural next phase of each schema inference method is *refactoring* or *refinement*, i.e. improving readability and simplifying structure while preserving the functionality of the resulting schema. Such requirement is ensured using a set of rules that enable one to describe equivalent or more general result. A demonstrative set of rules is depicted in Figure 19.

As we can see, the specified rules enable one to remove duplicate occurrence operators, to merge sequences of distinct occurrence operators into a single one, to merge

$a?? \Rightarrow a?$	$a^*? \Rightarrow a^*$	$aa^* \Rightarrow a^+$	$(ab) (ac) \Rightarrow a(b c)$
$a^{++} \Rightarrow a^+$	$a?^* \Rightarrow a^*$	$a^+a^* \Rightarrow a^+$	
$a^{**} \Rightarrow a^*$	$a^{+*} \Rightarrow a^*$	$a?a^+ \Rightarrow a^*$	
$a^{*+} \Rightarrow a^*$			
$a?^+ \Rightarrow a^*$			
$a^+? \Rightarrow a^*$			

Fig. 19. Merging of operators during refactorization

sequences of the same fragments, to avoid nondeterministic content models etc. Almost all of the mentioned approaches involve such a phase which indicates that the merging rules are not optimal and the approaches can still be improved. On the other hand, naturally, we can apply only those rules that do not interfere with other approaches, such as exploitation of splitting repetitions described in Section 4.3.7.

In phase III. (see Section 4.3) we have also discussed our extension Mlýnková (2009) of usage an obsolete schema. Apparently, using this approach, there may occur schema fragments that are not used in any of the input data. In other words, the input XML trees in I are valid against G_I^{opt} , however, it is too general and may cover also too distinct data. Hence, we extend the refactorization process with the following steps:

1. Pruning of unused schema fragments
2. Correction of lower and upper bounds of occurrences of schema fragments
3. Correction of operators

All the three steps can be done using a single linear passing of the input documents in I and preserving respective flags for particular schema parts.

4.5. Phase V. Inference of Simple Data Types Since most of the current approaches focus on inference of DTDs, they treat all text values as simple strings, i.e. they use common PCDATA data type. However, in XML Schema, as well as RELAX NG, we have a hierarchy of simple data types and also an option to specify user-defined types. Surprisingly, the inference problem of simple data types is currently highly marginalized. There seem to exist only two exceptions Chidlovskii (2002); Hegewald *et al.* (2006) which utilize the same approach: Each set of values of an element/attribute is simply analyzed to identify the minimal data type which contains all of them. Nevertheless, the authors focus only on numeric data types (such as `decimal`, `float`, `long`, `negativeInteger`), `date`, `binary` and `string`. Broader sets of simple types or even user-defined simple types are not supported so far.

4.6. Phase VI. Inference of Integrity Constraints Similarly to the case of inference of simple data types, the process of inference of *integrity constraints (ICs)* can be considered as an additional phase that can extend any of the inference strategies in general. An IC is a condition specified on the XML data. From this point of view we can consider simple data types as ICs as well. However, in this section we mean the “classical” ICs, in particular those that can be expressed in current XML schema languages (see Section 2). The most common type of ICs are keys and feign keys. As we have mentioned, in DTD they are expressed using simple data types `ID` and `IDREF(S)` and

are valid in the context of the whole document. In XML Schema we are provided with constructs `unique`, `key`, and `keyref` which enable one to specify the context/scope of the constraint. Apart from that, XML Schema involves new constructs `assert` and `report` that correspond to Schematron rules. However, none of the current approaches focuses on these advanced constructs.

Basic foundations and classifications of XML keys and discussion of the related decision problems can be found in Buneman *et al.* (2003), satisfiability of specification of keys is studied in Arenas *et al.* (2002).

Definition 4.10. A key is a construct $(C, P, \{L_1, L_2, \dots, L_k\})$, where $C, P, L_1, L_2, \dots, L_k$ are XPath paths without predicates that use only child and descendant axes. C is called context path, P target path and L_1, L_2, \dots, L_k key paths. C can be omitted, i.e. we can write $(P, \{L_1, L_2, \dots, L_k\})$. This is equivalent to $(/, P, \{L_1, L_2, \dots, L_k\})$. If C is omitted we call the key global key, otherwise, it is called relative key.

For the sake of simplicity we can consider that $k = 1$, i.e. a key is a construct $(C, P, \{L\})$. A key specifies the following condition: Let c be an element targeted by C and p and p' be two elements targeted by P from c . If the value targeted by L from p equals to the value targeted by L from p' , then p and p' are the same elements. In other words, no two different elements targeted by P from c can have the same value of L . This formalism corresponds to XML Schema keys. C specifies context elements, P target elements (`selector`), and L key elements (`field`).

Example 4.20. A key $(//project, member, \{mid\})$ formalizes the XML Schema key example depicted in Figure 20.

```

<element name="project">
  <!-- ... -->
  <key name="MemberKey">
    <selector path="member"/>
    <field path="mid"/>
  </key>
  <keyref name="MemberKeyRef" refer="MemberKey">
    <selector path="document"/>
    <field path="mid"/>
  </keyref>
</element>
    
```

Fig. 20. An example of XML Schema key and keyref constructs

Since the authors of Buneman *et al.* (2003) do not provide a formalism for foreign keys, we extend it in a similar manner.

Definition 4.11. A foreign key is a construct $(C, (P^1, \{L_1^1, L_2^1, \dots, L_k^1\}) \Rightarrow (P^2, \{L_1^2, L_2^2, \dots, L_k^2\}))$, where $(C, P^2, \{L_1^2, L_2^2, \dots, L_k^2\})$ is a key and $P^1, L_1^1, L_2^1, \dots, L_k^1$ are XPath paths without predicates that use only child and descendant axes. C can be omitted as in the case of keys.

For the sake of simplicity we can again consider that $k = 1$, i.e. a foreign key is a construct $(C, (P^1, \{L^1\}) \Rightarrow (P^2, \{L^2\}))$. Let c be an element targeted by C and p_1 be

an element targeted by P^1 from c . The foreign key specifies that there is an element p_2 targeted by P^2 from c such that the value targeted by L^1 from p_1 equals to the value targeted by L^2 from p_2 . In other words, each element targeted by P^1 from c refers to an element targeted by P^2 from c via the pair L^1 and L^2 . A foreign key in this formalism corresponds to XML Schema foreign keys. $(C, P^2, \{L^2\})$ is the referenced key (*refer*), C specifies context elements, P^1 target elements (*selector*), and L^1 foreign key elements (*field*).

Example 4.21. A foreign key $(//project, (document, \{mid\}) \Rightarrow (member, \{mid\}))$ formalizes the XML Schema foreign key (*keyref* construct) depicted in Figure 20.

Probably the first approach that enables one to search for XML keys can be found in Grahne and Zhu (2002). The authors first show that the set of candidate keys, i.e. sets of values that fulfill the condition of uniqueness in a specific context (see Definition 4.10), is large and we need to select the optimal one. They propose an algorithm based on a classical data-mining technique called *Apriori* which enables one to mine all frequent item sets. For finding *minimal* keys, i.e. the set where no key is inferable from others the authors exploit a sound and complete set of inference rules proposed in Buneman *et al.* (2003), such as, e.g.

$$\begin{aligned} (C, (P, S)) \wedge C' \subseteq C &\rightarrow (C', (P, S)) \\ (C, (P, S)) \wedge P' \subseteq P &\rightarrow (C, (P', S)) \end{aligned} \quad (11)$$

Another aim of the authors is to find so-called *approximate* keys, i.e. those valid in “almost” the whole XML document. For this purpose they introduce two concepts – *support* and *confidence* of key expression, i.e. measures of interestingness of a key expression from the point of view of the input data.

In Barbosa and Mendelzon (2003) the authors focus on the problem of inference of *ID* and also *IDREF(S)* attributes. In case of keys, they focus on the same issue, i.e. to identify the optimal key from the set of candidates, in this case using a greedy search strategy. Then, having a set of keys, the finding of foreign keys means just checking the condition specified by Definition 4.11.

In Nečaský and Mlýnková (2009) we propose an approach which enables to discover keys and foreign keys more precisely using the analysis of XML queries, in particular join queries. Assume a query Q which joins a sequence of elements S_1 targeted by a path P_1 with a sequence of elements S_2 targeted by a path P_2 on a condition $L_1 = L_2$. It means that Q joins an element e_1 from S_1 with an element e_2 from S_2 if e_1/L_1 equals to e_2/L_2 .

Example 4.22. An example of such query is depicted in Figure 21. It joins a sequence of elements targeted by a path $//employee$ with a list of elements targeted by a path $//member$ on a condition $eid = mid$ at line 05, i.e. an $//employee$ element e is joined with a $//member$ element m if eid of e equals to mid of m .

```

01 for $e in //employee
02 return
03 <employee>
04   {$e/name}
05   {for $m in //member[mid=$e/eid]
06     return
07       <member>{$m/./code,$m/position}</member>}
08   {let $d := //document[mid=$e/eid]
09     return
10       <doccnt>{count($d)}</doccnt>
11       <docavgpages>{avg($d/pages)}</docavgpages>}
12 </employee>

```

Fig. 21. Query with repeating join pattern

Assume that each join is done via a key/foreign key pair. Hence, we can infer from Q that L_1 is a key for elements in S_1 or L_2 is a key for elements in S_2 and the other is a foreign key referencing the key. From our sample query in Figure 21, we can therefore infer $(//employee, \{eid\})$ or $(//member, \{mid\})$. We can also infer the respective foreign key, i.e. $(//member, \{mid\}) \Rightarrow (//employee, \{eid\})$ or $(//employee, \{eid\}) \Rightarrow (//member, \{mid\})$, respectively.

The problem is how to decide which of L_1 and L_2 is the key and which is the foreign key. For this purpose we analyze the constructs used in the query (e.g. `for` vs. `let` clauses, aggregation functions such as `avg`, `min`, `max` or `sum`, function `count` etc.). On the basis of their usage we are able to output a list of scored keys and for each key a list foreign keys referencing the key. The score of a key can be negative or positive. A negative score means that the key is not specified by the XML documents while positive means that it is satisfied. The absolute value of the score means how sure we are about it. The method is supposed to be used in combination with any of the methods which analyze the data to optimize the search process.

4.7. Phase VII. Expressing the Inferred Items in the Target Language Last but not least we need to express the inferred schema using constructs of the target XML schema language. In case of DTD constructs it is a quite straightforward process since the inferred REs only need to be directly rewritten into the respective syntax using classical approaches such as a *state removal method* Du and Ko (2001); Linz (2000) or an *algebraic method* Brzozowski (1964); Kain (1972). Note that a crucial issue of these methods is to select the best order of states to remove, since different removal sequences lead to different regular expressions. For this purpose several heuristics are proposed in Han and Wood (2007).

In case of XML Schema constructs the situation is much more complicated due to extensions we proposed Vyhnanovská and Mlýnková (2010); Mlýnková and Nečaský (2009) and described in phase III. (see Section 4.3). In general in all cases we can exploit

classes C_{El} , C_{ElGr} and C_{At} (see Table 1). Depending on the type of the outlined production rule, we simply define respective globally defined element, group of elements, attribute or group of attributes. The problem is when we want use also classes C_{CT} and C_{CTDer} , i.e. globally defined complex types and their mutual derivation. Firstly, we naturally need to follow the W3C specifications Thompson *et al.* (2004). In particular, we have two choices – extension and restriction. Consequently, we can exploit outlining of complex types and their derivation only in case it can be specified in either of the two ways. Otherwise we must use its combination with classes C_{El} , C_{ElGr} and C_{At} . In addition, in general we usually have multiple options how to define the type hierarchy.

Example 4.23. Consider the example in Figure 22, where \vec{P} and \vec{Q} denote production rules outlined from original production rules \vec{M} and \vec{N} . In case (a) we exploit class C_{ElGr} and define groups P and Q that are referenced from element definitions of m and n . In case (b) we combine classes C_{ElGr} , C_{CT} and C_{CTDer} and define complex type P to be the ancestor of complex types of both elements m and n . Similarly, in case (c) Note that the authors of Buneman *et al.* (2003) define keys with more key paths.) we define content model of m , i.e. type mT , to be the ancestor of type nT derived by restriction.

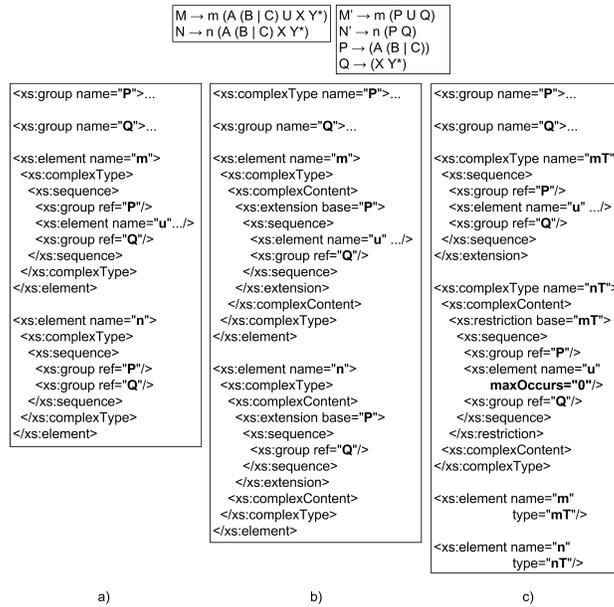


Fig. 22. Options of rewriting into XSD syntax

For selecting the optimal choice we again exploit a thesaurus, but this time we do not determine semantic similarity of element names, but their mutual hierarchy. The rules are relatively simple:

- Relationships *Broader Term/Narrower Term*, which specify more/less general terms, determine the broader term to be the candidate for ancestor.
- Relationships *Use/Used For*, which specify authorized/unauthorized terms, determine any of the terms to be the candidate for ancestor.
- Relationship *Related Term* determines related terms, for whom a common ancestor is created.
- In other cases, i.e. when the terms are not related or we cannot say anything about their semantic relationship, we do not exploit derived complex types.

Example 4.24. Consider again the example in Figure 22. If we knew that “*m*” is broader term of “*n*” (e.g. “*employee*” and “*director*”), we would choose schema *c*). If we knew that “*m*” and “*n*” are related terms (e.g. “*cat*” and “*dog*”), we would choose schema *b*). And if we knew that “*m*” and “*n*” are not related, but their content models are so similar that they were selected for merging, we would choose schema *a*).

If we consider the resulting schema of existing approaches, i.e. two element definitions that involve separate local specifications of their content models, we can see that any of the three results is much more informative and precise.

4.8. User Interaction Most of the previously described phases of general inference algorithm can be further optimized with the usage of user interaction (UI). The optimal situation would be if a user directly specified, e.g. the clusters of IG, the required merging rules, the outlined automata, target XML Schema constructs etc. However, since our primary aim is automatic inference, we cannot expect the user to execute the merging process, but to help it. Another problem is that the amount of user decisions cannot be too high, otherwise no one would exploit it.

In Vyhnanovská and Mlýnková (2010) we have proposed several cases when the inference process can benefit from user interaction. In particular we discuss the situations when the user can confirm/reject:

- clusters of IG to be merged, including clusters of elements with different names (phase II),
- subautomata to be outlined (phase III),
- candidates for unordered sequences (phase III), and
- XML Schema constructs (i.e. type inheritance and shared fragments) to be used in the target schema (phase VII).

Since all the cases are based on exploitation of a kind of similarity measure, we exploit a simple and straightforward two-threshold approach: If the particular similarity falls below $threshold_1$, the candidate is rejected automatically. If the distance falls between $threshold_1$ and $threshold_2$, UI is required. And if the result of similarity measure is higher than $threshold_2$, the candidate is automatically confirmed.

Last but not least, for the sake of clarity, the key characteristics of the described approaches are summed up in Table 3.

Table 3. Key characteristics of heuristic methods

Paper	Schema	Key Contributions
Shafer (1995); Moh <i>et al.</i> (2000)	SGML DTD	first simple heuristic merging rules
Garofalakis <i>et al.</i> (2000)	DTD	set of candidate solutions, MDL principle
Wong and Sankey (2003)	DTD	<i>sk</i> -string merging rule, ACO heuristics
Ahonen (1996)	DTD	<i>k, h</i> -context merging rule
Chidlovskii (2002); Hegewald <i>et al.</i> (2006)	XSD	precise occurrence ranges, simple data types
Vošta <i>et al.</i> (2008)	XSD	unordered sequences, STTGs and RTGs
Vyhnanovská and Mlýnková (2010)	XSD	user interaction
Mlýnková (2009)	DTD, XSD	exploitation of an obsolete schema
Mlýnková and Nečáský (2009)	XSD	globally defined schema fragments, exploitation of data semantics and data statistics
Grahne and Zhu (2002)	DTD	ID attributes
Barbosa and Mendelzon (2003)	DTD	ID, IDREF (S) attributes
Nečáský and Mlýnková (2009)	XSD	confirmation/rejection of keys/foreign keys using XML queries

5. Open Issues Although each of the existing approaches brings certain interesting ideas and optimizations, there is still a space of possible future improvements. We describe and discuss them in this section.

User Interaction In most of the existing papers the approaches focus on purely automatic inference of an XML schema. The problem is that the resulting schema may be highly unnatural. Although, e.g., the MDL principle evaluates the quality of the schema using a realistic assumption that it should tightly represent the data and, at the same time, be concise and compact, user preferences can be quite different. (Note that this is not the same motivation as in case of papers Bex *et al.* (2006, 2007) that focus on real-world DTDs and XSDs.) Hence, a natural improvement may be exploitation of user interaction. Some of the existing papers (e.g. Ahonen (1996)) mention the aspect of user interaction, typically in phase IV. (see Section 4.4) of refinement of the result, but there seems to be no detailed study and, in particular, respective implementation. And, naturally, this problem is closely related to a suitable user interface which does not require complex operations and decisions.

In paper Vyhnanovská and Mlýnková (2010) we have proposed our preliminary attempts towards exploitation of user interaction, however we can certainly go even further. For instance, the user may influence the merging phase by proposing preferred merging operations/target constructs, clustering similar elements etc. Such approach will not only enable one to find more concise result, but to find it more efficiently as well.

Other Input Information In all the existing works the XML schema is inferred on the basis of a set of positive examples, i.e. XML documents that should be valid against the inferred schema. As we have mentioned, the Gold's theorem highly restricts

the existing solutions and, hence, the authors focus on heuristic approaches or limit the methods to particular identifiable classes of languages. But another natural solution to the problem is to exploit additional information, such as an XML schema or XML queries.

In Mlýnková (2009) we have proposed a preliminary solution exploiting an obsolete schema, but the exploitation strategy can go even further. A related problem is being currently solved and, hence, an inspiration can be found in the area of *schema evolution* Guerrini and Mesiti (2008); Nečaský *et al.* (2009) or correction of XML data Bouchou *et al.* (2006); Staworko and Chomicki (2006); Svoboda and Mlýnková (2011) at a much sophisticated level.

In case of exploitation of XML queries the motivation is similar though more obvious. In Nečaský and Mlýnková (2009) we exploited XML queries just for the purpose of inference of integrity constraints. But, in general, the queries restrict parts of the data structure (those that should appear at output) and this partial information can be exploited for schema inference. A related problem is being solved in the area of *XML views* Papakonstantinou and Vianu (2000).

In addition, there seems to be no approach that would exploit negative examples (i.e. XML documents that should not conform to the schema). In this case we can find a real-world motivation again in the area of data evolution and versioning.

XML Schema Simple Data Types One of the biggest advantages of the XML Schema language in comparison to DTD is its wide support of simple data types Biron and Malhotra (2004). It involves 44 built-in data types such as, e.g., `string`, `integer`, `date` etc., as well as user-defined data types derived from existing simple types using `simpleType` construct. It enables one to derive new data types using *restriction* of values of an existing type (e.g. a string value having length greater than two), *list* of values of an existing type (e.g. list of integer values) or *union* of values of existing data types (e.g. union of positive and negative integers). Hence, a natural improvement of the existing approaches is a precise inference of simple data types. Unfortunately, most of the existing approaches omit the simple data types and consider all the values as strings. As we have mentioned, two exceptions are proposed in Chidlovskii (2002); Hegewald *et al.* (2006), but both the algorithms focus only on selected built-in data types.

Also note that the necessity to infer simple data types is naturally closely related to the purpose the schema is inferred for. Assuming that the resulting XML schema is used within a kind of XML data editor, the inferring module should propose also simple data types. On the other hand, if the inferred XML schema is used as a solution for approaches based on existence of an XML schema, e.g. schema-driven XML-to-relational mapping methods Shanmugasundaram *et al.* (1999); Mlýnková (2007), the simple data types are of marginal importance and, thus, can be omitted.

XML Schema Advanced Constructs The second big advantage of the XML Schema language are various complex constructs. The language exploits object-oriented features, such as user-defined data types, inheritance, polymorphism, i.e. substitutability of both data types and elements etc. Although most of these constructs do not extend the expressive power of XML Schema in comparison to DTD Mlýnková (2008b), they enable one to specify more user-friendly and, hence, realistic schemas. Naturally, their

usage is closely related to the previously described problem of user-interaction, since the user can specify which of the constructs are preferred. In Mlýnková and Nečaský (2009); Vošta *et al.* (2008); Vyhnanovská and Mlýnková (2010) we have proposed several preliminary approaches towards inference of unordered sequences, shared fragments or type inference. But, the language itself provides much stronger tools.

Integrity Constraints As we have mentioned, both DTD and XML Schema enable one to specify not only the structure of the data, but also various semantic constraints. Both involve `ID` and `IDREF(S)` data types that specify unique identifiers and references to them. The XML Schema language extends this feature using `unique`, `key` and `keyref` constructs that have the same purpose but enable one to specify the unique/key values more precisely, i.e. for selected subsets of elements and/or attributes and valid within a specified area. In addition, the `assert` and `report` constructs enable one to express specific constraints on values using the XPath language. The current works focus mainly on the `ID`, `IDREF(S)` attributes Grahne and Zhu (2002); Barbosa and Mendelzon (2003) and exploit various data mining approaches to find the optimal sets of keys and foreign keys. In Nečaský and Mlýnková (2009) we optimize the search strategy using analysis of XML queries. Unfortunately, all the existing works infer the keys separately, i.e. regardless a possibly existing XML schema or on the basis of an inference approach. Similarly, none of them focusses on any of the advanced constraints of XML Schema or Schematron. In addition, there are also more complex XML integrity constraints Opočenská and Kopecký (2008) that could be inferred, though they cannot be expressed in the existing schema specification languages so far, functional dependencies Yu and Jagadish (2008); Fassetti and Fazzinga (2007) or even languages for expressing any integrity constraint in general, such as, e.g., *Object Constraint Language (OCL)* OMG (2009). A detailed study of XML integrity constraints can be found in Fan (2005), whereas their inference would extend the optimization of approaches that analyze and exploit information on XML data from XML schemas.

Other Schema Definition Languages The DTD and XML Schema are naturally not the only languages for definition of structure of XML data, though they are undoubtedly the most popular ones. The obvious reason is that these two have been proposed by the W3C, whereas DTD is even a part of specification of XML. Nevertheless, there are also other relatively popular schema specification languages, the two most popular ones, RELAX NG and Schematron, are briefly introduced in Section 2. The former language has higher expressive power than XML Schema and DTD, since it enables one, e.g., to combine elements and attributes in the regular expressions. The latter one exploits completely different approach (since it is a pattern-based, not grammar-based language) and, hence, it will require completely different inference approach.

Data Streams A special type of XML data that have only recently become popular and, hence, the necessity for proposing respective processing approaches is crucial are so-called *XML data streams*. In this particular application the input data are so huge and/or their amount is so high that they cannot be kept in a memory concurrently, they cannot be read more than once or their processing cannot “wait” for the last portion of the data. Hence the situation is much more complicated. All the XML technologies are currently

being accommodated to stream processing and it is only a matter of time when respective efficient schema inference approach will be required as well.

6. Summary and Future Work In general, the XML schema of XML documents is currently exploited mainly for two purposes – data-exchange and optimization. In the former case we usually need the inferred schema as a candidate schema further improved by a user using an appropriate editor, or in cases when no schema is available. In the latter case the approaches exploit the knowledge of the schema, i.e. the expected structure of the data, for optimization purposes such as, e.g., finding the optimal storage Shanmugasundaram *et al.* (1999) or compression Augeri *et al.* (2007) strategy. However, in general, almost any approach that deals with XML data can benefit from the knowledge of their structure, i.e. XML schema. The only question is to what extent.

In this paper we focussed on the most popular group of approaches for semi-automatic inference of XML schema for a given set of XML documents – heuristic methods. Their popularity is given by the fact that their key aim is to provide natural and realistic results, despite we cannot specify any special features of the resulting schemas.

The main contribution of this paper can be summed up as follows:

- A general framework that characterizes the classical phases of algorithm for heuristic inference of XML schemas.
- A study of current approaches and their improvements of the particular phases of the general inference algorithm.
- A detailed description of several optimization approaches we proposed in recent years and their comparison with current approaches.
- A study of open issues to be solved in the area of schema inference in general.

Recently we have finished implementation of a general and extensible framework called *jInfer* Klempa *et al.* (2012) that covers the general inference phases. Using plugins it enables one to change the respective approaches and compare their influence on the inference process as well as results in general. (Note that a similar system, called *SchemaScope*, focussing on the grammar-inferring approaches, was described in Bex *et al.* (2008).) In the next phase we will implement the current approaches and provide their detailed analysis and comparison using nontrivial set of both real-world and synthetic data. And, finally, in our future work we will focus mainly on the open issues stated in Section 5. Our primary aim is to study the advantages of other input information (such as XML queries or XML operations in general) and to infer broader information on the data, in particular integrity constraints. Our preliminary results in this area using *jInfer* can be found in Švirec and Mlýnková (2012); Klempa *et al.* (2012b); Vitásek and Mlýnková (2012).

References

- Ahonen, H. (1996). *Generating Grammars for Structured Documents Using Grammatical Inference Methods*. Report A-1996-4, Department of Computer Science, University of Helsinki.

- Arenas, M., Fan, W., Libkin, L. (2002). On Verifying Consistency of XML Specifications. In: *Proceedings of PODS '02*. ACM Press, Madison, Wisconsin, USA, pp. 259–270.
- Augeri, C. J., Bulutoglu, D. A., Mullins, B. E., Baldwin, R. O., Baird, III, Leemon C. (2007). An Analysis of XML Compression Efficiency. In: *Proceedings of ExpCS '07*, article no. 7. ACM Press, San Diego, California, USA.
- Barbosa, D., Mendelzon, A. (2003). Finding ID Attributes in XML Documents. In: *Proceedings of Xsym 2003*. Lecture Notes in Computer Science, vol. 2824. Springer Verlag, Berlin / Heidelberg, pp. 180–194.
- Barták, R. (1998). *On-Line Guide to Constraint Programming*. <http://kti.mff.cuni.cz/~bartak/constraints/>.
- Bex, G. J., Neven, F., Van den Bussche, J. (2004). DTDs versus XML Schema: a Practical Study. In: *Proceedings of WebDB '04*. ACM Press, New York, NY, USA, pp. 79–84.
- Bex, G. J., Neven, F., Schwentick, T., Tuyls, K. (2006). Inference of Concise DTDs from XML Data. In: *Proceedings of VLDB '06*. VLDB Endowment, Seoul, Korea, pp. 115–126.
- Bex, G. J., Neven, F., Vansummeren, S. (2007). Inferring XML Schema Definitions from XML Data. In: *Proceedings of VLDB '07*. VLDB Endowment, Vienna, Austria, pp. 998–1009.
- Bex, G. J., Neven, F., Vansummeren, S. (2008). SchemaScope: a System for Inferring and Cleaning XML Schemas. In: *Proceedings of SIGMOD '08*. ACM Press, Vancouver, Canada, pp. 1259–1262.
- Biron, P. V., Malhotra, A. (2004). *XML Schema Part 2: Datatypes (Second Edition)*. W3C. <http://www.w3.org/TR/xmlschema-2/>.
- Bouchou, B., Cheriati, A., Alves, M. H. ., Savary, A. (2006). Integrating Correction into Incremental Validation. In: *Informal Proceedings of BDA '06*. Lille, France.
- Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F. (2008). *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C. <http://www.w3.org/TR/REC-xml>.
- Brzozowski, J. A. (1964). Derivatives of Regular Expressions. *J. ACM*, **11**(4), 481–494.
- Buneman, P., Davidson, S., Fan, W., Hara, C., Tan, W.-C. (2003). Reasoning about Keys for XML. *Inf. Syst.*, **28**(8), 1037–1063.
- Chidlovskii, B. (2002). Schema Extraction from XML Collections. In: *Proceedings of JCDL '02*. ACM Press, Portland, Oregon, USA, pp. 291–292.
- Clark, J. (1999). *XSL Transformations (XSLT) Version 1.0*. W3C. <http://www.w3.org/TR/xslt>.

- Clark, J., DeRose, S. (1999). *XML Path Language (XPath) Version 1.0*. W3C. <http://www.w3.org/TR/xpath>.
- Dorigo, M., Birattari, M., Stutzle, T. (2006). *Ant Colony Optimization – Artificial Ants as a Computational Intelligence Technique*. Report TR/IRIDIA/2006-023, IRIDIA, Bruxelles, Belgium.
- Du, D.-Z., Ko, K.-I. (2001). *Problem Solving in Automata, Languages, and Complexity*. Wiley-Interscience; 1st edition.
- Du, F., Amer-Yahia, S., Freire, J. (2004). ShreX: Managing XML Documents in Relational Databases. In: *Proceedings of VLDB '04*. VLDB Endowment, Toronto, Canada, pp. 1297–1300.
- Fan, W. (2005). XML Constraints: Specification, Analysis, and Applications. In: *Proceedings of DEXA '05 Workshops*. IEEE Computer Society, Copenhagen, Denmark, pp. 805–809.
- Fassetti, F., Fazzinga, B. (2007). FOX: Inference of Approximate Functional Dependencies from XML Data. In: *Proceedings of DEXA '07*. IEEE Computer Society, Washington, DC, USA, pp. 10–14.
- Gao, S., Sperberg-McQueen, C. M., Thompson, H. S. (2009). *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. W3C. <http://www.w3.org/TR/xmlschema1-1/>.
- Garofalakis, M., Gionis, A., Rastogi, R., Seshadri, S., Shim, K. (2000). XTRACT: a System for Extracting Document Type Descriptors from XML Documents. *SIGMOD Rec.*, **29**(2), 165–176.
- Gold, E. M. (1967). Language Identification in the Limit. *Information and Control*, **10**(5), 447–474.
- Grahne, G., Zhu, J. (2002). Discovering Approximate Keys in XML Data. In: *Proceedings of CIKM '02*. ACM Press, McLean, Virginia, USA, pp. 453–460.
- Grunwald, P.D. (2005). *A Tutorial Introduction to the Minimum Description Principle*. Centrum voor Wiskunde en Informatica. <http://homepages.cwi.nl/~pdg/ftp/mdlintro.pdf>.
- Guerrini, G., Mesiti, M. (2008). X-Evolution: A Comprehensive Approach for XML Schema Evolution. In: *Proceedings of DEXA '08*. IEEE Computer Society, Washington, DC, USA, pp. 251–255.
- Han, Y.-O., Wood, D. (2007). Obtaining Shorter Regular Expressions from Finite-State Automata. *Theor. Comput. Sci.*, **370**(1-3), 110–120.

- Hegewald, J., Naumann, F., Weis, M. (2006). XStruct: Efficient Schema Extraction from Multiple and Large XML Documents. In: *Proceedings of ICDEW '06*. IEEE Computer Society, Washington, DC, USA, pp. 81–81.
- Jain, Anil K., Dubes, Richard C. (1988). *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- Jelliffe, R. (2001). *The Schematron – An XML Structure Validation Language using Patterns in Trees*. <http://xml.ascc.net/resource/schematron/>.
- Kain, R. Y. (1972). *Automata Theory: Machines and Languages*. McGraw-Hill Inc., USA.
- Klempa, M., Mikula, M., Smetana, R., Švirec, M., Vitásek, M. *jInfer XML Schema Inference Framework*. <http://jinfer.sourceforge.net/modules/paper.pdf>.
- Klempa, M., Stárka, J., Mlýnková, I. (2012). Optimization and Refinement of XML Schema Inference Approaches. In: *Proceedings of ANT '12*, vol. 10. Elsevier, Niagara Falls, Canada, pp. 120–127.
- Levenshtein, V. I. (1966). Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, **10**(8), 707–710.
- Linz, P. (2000). *An Introduction to Formal Languages and Automata*. Jones & Bartlett Publishers; 3rd edition.
- Mani, M. (2001). *Keeping Chess Alive: Do We Need 1-Unambiguous Content Models?* Montreal, Canada talk given at Extreme Markup Languages.
- Marian, A. (2002). Detecting Changes in XML Documents. In: *Proceedings of ICDE '02*. IEEE Computer Society, Washington, DC, USA, pp. 41–52.
- Mignet, L., Barbosa, D., Veltri, P. (2003). The XML Web: a First Study. In: *Proceedings of WWW '03*. ACM Press, Budapest, Hungary, pp. 500–510.
- Mlýnková, I. (2008a). An Analysis of Approaches to XML Schema Inference. In: *Proceedings of SITIS '08*. IEEE Computer Society, Los Alamitos, CA, USA, pp. 16–23.
- Mlýnková, I. (2009). On Inference of XML Schema with the Knowledge of an Obsolete One. In: *Proceedings of ADC '09*. Australian Computer Society, Inc., Darlinghurst, Australia, pp. 77–84.
- Mlýnková, I., Nečaský, M. (2009). Towards Inference of More Realistic XSDs. In: *Proceedings of SAC '09*. ACM Press, Honolulu, Hawaii, USA, pp. 639–646.
- Mlýnková, I., Toman, K., Pokorný, J. (2006). Statistical Analysis of Real XML Data Collections. *Proceedings of COMAD'06*. Tata McGraw-Hill, New Delhi, India, pp. 20–31.

- Mlýnková, I. (2007). A Journey towards More Efficient Processing of XML Data in (O)RDBMS. In: *Proceedings of CIT '07*. IEEE Computer Society, Washington, DC, USA, pp. 23–28.
- Mlýnková, I. (2008b). Similarity of XML Schema Definitions. *Proceeding of DocEng '08*. ACM Press, Sao Paulo, Brazil, pp. 187–190.
- Moh, C.-H., Lim, E.-P., Ng, W.-K. (2000). Re-engineering Structures from Web Documents. In: *Proceedings of DL '00*. ACM Press, San Antonio, Texas, USA, pp. 67–76.
- Murata, M. (2002). *RELAX (Regular Language Description for XML)*. <http://www.xml.gr.jp/relax/>.
- Murata, M., Lee, D., Mani, M., Kawaguchi, K. (2005). Taxonomy of XML Schema Languages Using Formal Language Theory. *ACM Trans. Internet Technol.*, 5(4), 660–704.
- Nečaský, M., Mlýnková, I. (2009). Discovering XML Keys and Foreign Keys in Queries. In: *Proceedings of SAC '09*. ACM Press, Honolulu, Hawaii, USA, pp. 632–638.
- Nečaský, M., Klímeck, J., Kopenec, L., Kučerová, L., Malý, J., Opočenská, K. (2009). *XCase – A Case Tool for Designing XML*. <http://www.codeplex.com/xcase>.
- Nierman, A., Jagadish, H. V. (2002). Evaluating Structural Similarity in XML Documents. In: *Proceedings of WebDB'02*. ACM Press, Madison, Wisconsin, USA, pp. 61–66.
- OMG. (2009). *Object Constraint Language Specification, version 2.0*. <http://www.omg.org/technology/documents/formal/ocl.htm>.
- Opočenská, K., Kopecký, M. (2008). Incox – a Language for XML Integrity Constraints Description. In: *Proceedings of DATESO'08*. CEUR-WS.org, Desna – Cerna Ricka, Czech Republic, pp. 1–12.
- Papakonstantinou, Y., Vianu, V. (2000). DTD Inference for Views of XML Data. In: *Proceedings of PODS '00*. ACM Press, Dallas, Texas, USA, pp. 35–46.
- Peterson, D., Biron, P. V., Malhotra, A., Sperberg-McQueen, C. M. (2009). *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. W3C. <http://www.w3.org/TR/xmlschema11-2/>.
- Shafer, K. E. (1995). Creating DTDs via the GB-Engine and Fred. In: *Proceedings of SGML'95*. Graphic Communications Association, pp. 399.
- Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D. J., Naughton, J. F. (1999). Relational Databases for Querying XML Documents: Limitations and Opportunities. In: *Proceedings of VLDB '99*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 302–314

- Staworko, S., Chomiccki, J. (2006). Validity-Sensitive Querying of XML Databases. In: *Proceedings of EDBT '06*. Lecture Notes in Computer Science, vol. 4254. Springer Verlag, Berlin / Heidelberg, pp. 164–177.
- Svoboda, M., Mlýnková, I. (2011). Correction of Invalid XML Documents. In: *Proceedings of DASFAA '11*. Springer Verlag, Hong Kong, China, pp. 179–194.
- Thompson, H. S., Beech, D., Maloney, M., Mendelsohn, N. (2004). *XML Schema Part 1: Structures (Second Edition)*. W3C. <http://www.w3.org/TR/xmlschema-1/>.
- Torsello, A., Hancock, E. R. (2003). Computing Approximate Tree Edit Distance Using Relaxation Labeling. *Pattern Recogn. Lett.*, **24**(8), 1089–1097.
- Touzet, H. (2003). Tree Edit Distance with Gaps. *Inf. Process. Lett.*, **85**(3), 123–129.
- Touzet, H. (2005). A Linear Tree Edit Distance Algorithm for Similar Ordered Trees. In: *Proceedings of CPM '05*. Lecture Notes in Computer Science. Springer Verlag, Heidelberg, pp. 334–345.
- Vitásek, M., Mlýnková, I. (2012). Inference of XML Integrity Constraints. In: *Proceedings of ADBIS '12*. Springer Verlag, Poznan, Poland (in press).
- Vošta, O., Mlýnková, I., Pokorný, J. (2008). Even an Ant Can Create an XSD. In: *Proceedings of DASFAA'08*. Springer Verlag, New Delhi, India, pp. 35–50.
- Švirec, M., Mlýnková, I. (2012). Efficient Detection of XML Integrity Constraints Violation. In: *Proceedings of NDT '12*. Springer Verlag, Dubai, UAE, pp. 259–273.
- Vyhnanovská, J., Mlýnková, I. (2010). Interactive Inference of XML Schemas. In: *Proceedings of RCIS '10*. IEEE Computer Society, Nice, France, pp. 191–202.
- Wong, R. K., Sankey, J. (2003). *On Structural Inference for XML Data*. Report UNSW-CSE-TR-0313, School of Computer Science, The University of New South Wales.
- Yu, C., Jagadish, H. V. (2008). XML Schema Refinement Through Redundancy Detection and Normalization. *The VLDB Journal*, **17**(2), 203–223.

I. Mlýnková received her Ph.D. degree in Computer Science in 2007 from the Charles University in Prague, Czech Republic. She is an assistant professor at the Department of Software Engineering of the Charles University and an external member of the Department of Computer Science and Engineering of the Czech Technical University. She has published more than 60 publications, 4 gained the Best Paper Award. She is a PC member or reviewer of 15 international events and co-organizer of 3 international workshops (X-Schemas@ADBIS, MoViX@DEXA, BenchmarX@DASF AA, all since 2009).

M. Nečaský received his Ph.D. degree in Computer Science in 2008 from the Charles University in Prague, Czech Republic, where he currently works in the Department of Software Engineering as an assistant professor. He is an external member of the Department of Computer Science and Engineering of the Faculty of Electrical Engineering, Czech Technical University in Prague. His research areas involve XML data design, integration and evolution. He is an organizer or PC chair of three international workshops. He has published 15 refereed conference papers (2 received the Best Paper Award). He has published 3 book chapters and a book.

Chapter 5

Structural and Semantic Aspects of Similarity of Document Type Definitions and XML Schemas

Aleš Wojnar
Irena Mlýnková
Jiří Dokulil

Published in the *Special Issue on Intelligent Distributed Information Systems* of the *International Journal on Information Sciences*, volume 180, issue 10, pages 1817–1836. Elsevier, May 2010. ISSN 0020-0255.

Impact Factor: 2.836
5-Year Impact Factor: 3.009





Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Structural and semantic aspects of similarity of Document Type Definitions and XML schemas

Aleš Wojnar, Irena Mlýnková*, Jiří Dokulil

Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Malostranské nám. 25, 118 00 Praha 1, Czech Republic

ARTICLE INFO

Article history:

Received 21 November 2008

Received in revised form 6 November 2009

Accepted 22 December 2009

Keywords:

XML schema

DTD

XSD

Similarity

Data semantics

Structural analysis

ABSTRACT

The natural optimization strategy for XML-to-relational mapping methods is exploitation of similarity of XML data. However, none of the current similarity evaluation approaches is suitable for this purpose. While the key emphasis is currently put on semantic similarity of XML data, the main aspect of XML-to-relational mapping methods is analysis of their structure.

In this paper we propose an approach that utilizes a verified strategy for structural similarity evaluation – tree edit distance – to DTD constructs. This approach is able to cope with the fact that DTDs involve several types of nodes and can form general graphs. In addition, it is optimized for the specific features of XML data and, if required, it enables one to exploit the semantics of element/attribute names. Using a set of experiments we show the impact of these extensions on similarity evaluation. And, finally, we discuss how this approach can be extended for XSDs, which involve plenty of “syntactic sugar”, i.e. constructs that are structurally or semantically equivalent.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

Without any doubt the eXtensible Markup Language (XML) [13] has become a standard for data representation and manipulation. This situation naturally initiated a boom in the efficient implementation of XML data management and processing tools. One of the typical optimization strategies is exploitation of similarity of XML data. In general it enables one to treat similar data in a similar way or to extend appropriate approaches verified for particular data to the whole set of similar ones.

In this paper we deal with similarity of XML schema fragments expressed in Document Type Definitions (DTDs) [13] or XML Schema definitions (XSDs) [62,11]. In this area we can deal with either *quantitative* or *qualitative* similarity measure. In the former case we are interested in the degree of difference of the schemas, in the latter one we also want to know how the schemas relate, e.g. which of the schemas is more general. In this paper we focus on quantitative measure which is currently used in a huge number of applications such as clustering of XML data [17,59], dissemination-based applications [6,68], schema integration systems [42,34] or other less obvious areas [53], such as, e.g., e-commerce, semantic and approximate query processing, etc.

In our case we focus on the exploitation of similarity in schema-driven XML-to-relational mapping strategies [44] where the similarity of schema fragments can be used for determining the optimal storage strategy in the case of fragments for which we do not have any other information. The problem we are facing is that none of the current approaches is suitable for this purpose, since the key emphasis is currently put on semantic similarity. However, since the key aspect of

* Corresponding author.

E-mail addresses: ales.wojnar@gmail.com (A. Wojnar), irena.mlynkova@mff.cuni.cz (I. Mlýnková), jiri.dokulil@mff.cuni.cz (J. Dokulil).

XML-to-relational storage strategies is the structure of schema fragments, we focus on more precise structural analysis. On the other hand, since the semantics of the data can also be important information, we still preserve the exploitation of semantic similarity. To fulfill both the aims we combine and adapt to DTD constructs two verified approaches – tree edit distance and semantics of element/attribute names. We show how a well-known and verified methodology of edit distance can be utilized for DTDs that can involve several types of nodes and form general graphs and even extended with exploitation of semantic similarity. Using a set of experiments we show the impact of these extensions on similarity evaluation. And, last but not least, we show how this approach can be extended for XSDs, which involve more complex structures than DTDs and, in particular, plenty of “syntactic sugar”, i.e. constructs that are structurally or semantically equivalent.

The paper is structured as follows: Section 2 provides the background information on related technologies, as well as an overview of terms used in the rest of the text. Section 3 reviews the related works dealing with similarity of XML data from various points of view. Section 4 describes the proposed approach in detail and Section 5 discusses the results of related experiments. In Section 6 we describe a possible extension of the proposal for XML Schema definitions. Section 7 discusses the main advantages and disadvantages of the algorithm and compares it with the current approaches. And, finally, Section 8 provides conclusions and outlines future work.

This paper is in fact an extended version of paper [66]. In addition to the strategies and algorithms included to that earlier work, we now briefly describe how the approach proposed for DTDs can be further extended for XSD-specific constructs. The full description of this approach can be found in paper [45].

2. Background

An XML document is usually viewed as a directed ordered labeled tree with several types of nodes whose edges represent relationships among them.

Definition 1. An XML document is a directed ordered labeled tree $T = (V, E, \Sigma_E, \Sigma_A, \Gamma, lab, r)$, where V is a finite set of nodes, $E \subseteq V \times V$ is a set of edges, Σ_E is a finite set of element names, Σ_A is a finite set of attribute names, Γ is a finite set of text values, $lab : V \rightarrow \Sigma_E \cup \Sigma_A \cup \Gamma$ is a surjective function which assigns a label to each $v \in V$, whereas v is an *element* if $lab(v) \in \Sigma_E$, an *attribute* if $lab(v) \in \Sigma_A$ or a *text value* if $lab(v) \in \Gamma$ and r is the root node of the tree.

An example of an XML document and its tree representation is depicted in Fig. 1.

The allowed structure of an XML document can be described using an XML schema. The most popular XML schema language is currently the Document Type Definition (DTD) [13]. A simple example is depicted in Fig. 2 on left-hand side.

The key aspect of a DTD is a description of the allowed structure of an element using its *content model*.

Definition 2. A content model α over a set of element names Σ'_E is a regular expression defined as $\alpha = \epsilon | pcdata | f | (\alpha_1, \alpha_2, \dots, \alpha_n) | (\alpha_1 | \alpha_2 | \dots | \alpha_n) | \beta^* | \beta^+ | \beta^?$, where ϵ denotes the empty content model, $pcdata$ denotes the text content, $f \in \Sigma'_E$, “,” and “|” stand for concatenation and union (of content models $\alpha_1, \alpha_2, \dots, \alpha_n$) and “*”, “+” and “?” stand for zero or more, one or more and optional occurrence(s) (of content model β), respectively.

Definition 3. An XML schema S is a four-tuple $(\Sigma'_E, \Sigma'_A, \Delta, s)$, where Σ'_E is a finite set of element names, Σ'_A is a finite set of attribute names, Δ is a finite set of declarations of the form $e \rightarrow \alpha$ or $e \rightarrow \beta$, where $e \in \Sigma'_E$, α is a content model over Σ'_E and $\beta \subseteq \Sigma'_A$ and $s \in \Sigma'_E$ is a start symbol.

To simplify the processing, an XML schema is often transformed into a graph representation called *DTD graph* [56]. An example of a schema graph is depicted in Fig. 2 on right-hand side.

Definition 4. A schema graph of a schema $S = (\Sigma'_E, \Sigma'_A, \Delta, s)$ is a directed, labeled graph $G = (V, E, lab')$, where V is a finite set of nodes, $E \subseteq V \times V$ is a set of edges, $lab' : V \rightarrow \Sigma'_E \cup \Sigma'_A \cup \{“|”, “*”, “+”, “?”, “,”, “|”\} \cup \{pcdata\}$ is a surjective function which assigns a label to $\forall v \in V$ and s is the root node of the graph.

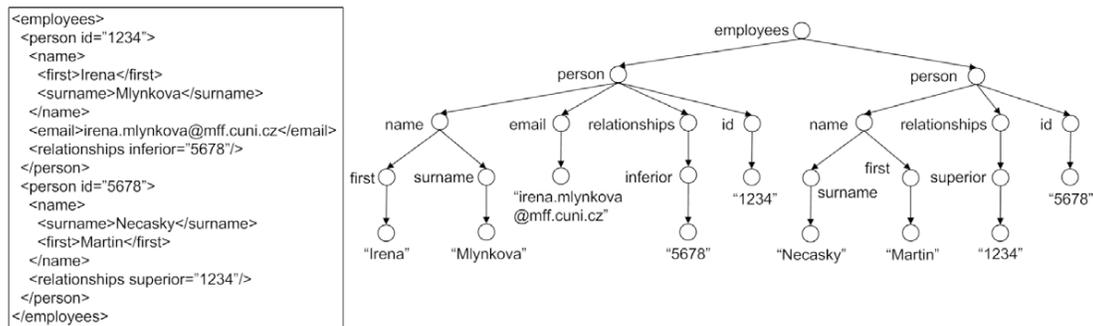


Fig. 1. An example of an XML document and its tree representation.

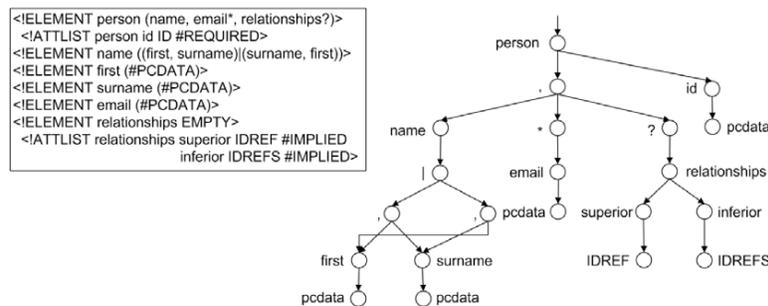


Fig. 2. An example of a DTD and its graph representation.

3. Related work

The number of current papers in the area of XML data similarity is huge. We can evaluate similarity among XML documents, XML schemas or between the two groups. We can distinguish several levels of similarity, such as, e.g., structural level, semantic level or constraint level. Or we can require different precisions of the similarity.

3.1. Similarity of XML documents

In the case of document similarity the amount of related techniques is high. In general, we distinguish techniques expressing similarity of two documents D_x and D_y using the edit distance of their tree representations T_x and T_y (e.g. [50,18,16,32]) and techniques which specify a simpler representation of D_x and D_y that enables more efficient similarity evaluation (e.g. [52,71,26,64]).

The edit distance strategy was originally used for comparing similarity between two strings s_x and s_y [28,35]. It is based on the idea of finding the cheapest sequence of edit operations that can transform s_x into s_y . The edit operations can be defined variously. For example, paper [28] uses a single operation – *substitution* of a single character, therefore this algorithm can be used only for strings of the same length. On the other hand, the Levenshtein distance algorithm [35] supports *insertion*, *deletion* and *substitution* of a single character. In addition, a non-negative constant cost is associated with each operation. In the simplest case all the operations cost one unit except for substitution of identical characters, in which case the cost is zero.

The edit distance can be applied on labeled trees as well, only the set of operations needs to be adapted respectively. The basic set of tree edit operations involves *insertion* and *deletion* of a single node and *relabeling* of a node. However, the respective approaches [60,65,15] are appropriate only for general trees, not for XML data structure. The problem is that as two documents created from the same DTD can have radically different structure (due to repeatable, optional and alternative elements), they would compute an undesirably high distance. Consequently, the true XML tree edit strategies [50,18,16,32,19] involve more complex operations, in particular *insertion* and *deletion* of a whole subtree. In general, the approaches are quite similar; they slightly differ only in the sets of simple operations (such as, e.g., inserting/deleting a leaf or inner node, moving a node, etc.) and the way they search for the shortest sequence of edit operations.

The idea of the latter set of approaches [52,71,26,47,41] is that it is possible to find a simpler representation of XML documents that enables one to evaluate their similarity more efficiently. The basic approaches represent an XML document using a set of elements, root paths or even all paths (possibly with respective frequencies) and, hence, reduce the problem to similarity of the sets. Paper [26] proposes an interesting approach that is based on the idea of representing an XML document as a time series in which each occurrence of a (start or end) tag represents an impulse. Hence, it reduces the problem to similarity of two signals. However, the main disadvantage of all these approaches is that they do not consider all XML features such as, e.g., the order of elements.

3.2. Similarity of an XML document and an XML schema

In the case of the similarity of a document D and a schema S we cannot transform the problem to measuring similarity of two ordered labeled trees, since, on the one hand, we have a tree, but we have to match it with a schema, i.e. a set of regular expressions. Thus, the problem is much complicated and the set of solutions is quite small.

We can again identify two types of strategies. On one hand, there are techniques [9,10] based on the fact that while matching D against S , some attributes and subelements of an element in S can be missing from the corresponding element in D or D can contain some additional attributes and/or subelements. Hence, they measure the number of elements which appear in D but not in S and vice versa. On the other hand, there are techniques [49,14] that measure the closest distance between D and “all” documents valid against S . The idea is relatively simple: Since an XML schema can be described as an *extended context free grammar* [8], S is represented using an automaton (built using Thompson’s classical algorithm [63]) and the problem is reduced to evaluation of the edit distance between the automaton and a document tree.

3.3. Similarity of XML schemas

In the case of methods for measuring the similarity of two XML schemas S_x and S_y , we consider the problem of similarity of two sets of regular expressions, typically called a *schema matching problem* [57]. In comparison with the previous case the amount of current techniques is enormous.

The vast majority of the approaches exploit various supplemental *matchers* [53], i.e. functions which evaluate similarity of a particular feature of the given schema fragments, such as, e.g., similarity of labels of leaf nodes or root nodes [40], similarity of context [38,39,34,54,69] or paths [70,61], similarity of schema instances [37], etc. Some of the approaches also exploit additional information, such as predefined similarity rules [42], previous results [37], user interaction [21], machine-learning strategies [37,31], knowledge of related queries [25], constraints [55], etc. All the partial results are then combined into the resulting similarity value, usually using a kind of a weighted sum, either simple or advanced [23]. Some of the approaches also focus on special situations, such as matching large schemas [22], matching large number of schemas [59] or matching several schemas at the same time [30]. But, in general, with regard to the purpose of the similarity measure, the approaches focus mostly on semantic aspects of data, i.e. similarity of element/attribute names.

A special type of approach has only recently been proposed in [5]. Similarly to the case of XML documents, it is based on the idea of finding a simpler representation of XML schema fragments that enables one to evaluate their similarity more efficiently. In this particular case the authors exploit Prufer sequences.

In general, due to various approximations and simplifications, the approaches provide more or less precise results. Therefore, a new research area has recently opened dealing with verification of the resulting matches, i.e. schema fragments denoted as similar [12]. However, these approaches are already beyond the scope of this paper.

4. Proposed algorithm

As we have already described in the introduction, we exploit similarity of XML schema fragments in order to allow optimization of XML-to-relational mapping strategies [44]. Hence, contrary to current approaches which search for matching between semantically related schema fragments, we search for schema fragments which are related structurally. However, as the semantics of element/attribute names is an important aspect, we want to involve this information as well.

The algorithm we propose is based mainly on the work presented in [50] which focuses on expressing the similarity of XML documents D_x and D_y using tree edit distance. The main contribution of the algorithm is in introducing two new edit operations *InsertTree* and *DeleteTree* which allow the manipulation of more complex structures than a single node. Since repeated structures can be found in a DTD as well, if it contains shared or recursive elements, we must also involve these operations. However, contrary to XML documents that can be modeled as trees, DTDs can, in general, form general cyclic graphs. Thus, procedures for computing edit distance of trees need to be modified for use with DTD graphs.

The proposed approach can be divided into three parts as depicted in Algorithm 1. The input DTDs are firstly parsed (lines 1 and 2) and their tree representations are constructed. Next, costs for tree inserting (line 3) and tree deleting (line 4) are computed. In particular, we pre-compute the cost for deleting all subtrees of T_x and inserting all subtrees of T_y . And, finally (line 5), we compute the resulting edit distance, i.e. similarity.

Algorithm 1. Main body of the proposed approach

Input: DTD_x, DTD_y

Output: Edit distance between DTD_x and DTD_y

1: $T_x = \text{ParseXSD}(DTD_x)$;

2: $T_y = \text{ParseXSD}(DTD_y)$;

3: $\text{Cost}_{\text{Graft}} = \text{ComputeCost}(T_y)$;

4: $\text{Cost}_{\text{Prune}} = \text{ComputeCost}(T_x)$;

5: **return** $\text{EditDistance}(T_x, T_y, \text{Cost}_{\text{Prune}}, \text{Cost}_{\text{Graft}})$;

4.1. DTD Tree construction

The key operation of our approach is tree representation of the given DTDs. As we have mentioned, the problem is that a DTD graph can contain both undirected and directed cycles. In addition, the structure of a DTD can be quite complex – the specified content models can contain arbitrary combinations of operators (i.e. “|” or “,”) and cardinality constraints (i.e. “?”, “*” or “+”). Therefore, we must first simplify the complex regular expressions using a set of transformation rules and we eliminate the cycles.

4.1.1. Simplification of DTDs

In order to simplify content models we use a classical approach – a set of transformation rules. The biggest set was probably defined in [56], but these simplifications are too strong for our purpose. Hence, we use only a subset of them as depicted in Figs. 3 and 4.

The rules enable one to convert all element definitions so that each cardinality constraint operator is connected to a single element and, hence, we can merge the nodes of an element and a respective operator in the graph representation. The second purpose is to avoid usage of “|” operator. Note that some of the rules do not produce equivalent XML schemes and cause a kind of information loss. But this aspect is common for all current XML schema similarity measures – it seems that the full generality of the regular expressions cannot be captured easily. On the other hand, for the purpose of XML-to-relational mapping, this loss of information is acceptable since we are usually interested in general aspects such as multiple or optional occurrence of an element.

4.1.2. Shared and recursive elements

In this step we need to eliminate shared and recursive elements, i.e. undirected and directed cycles in the DTD graph. In the case of a shared element the solution is simple: We create its separate copy for each sharer as depicted in Fig. 5.

The problem is that in the case of recursive elements the same idea would invoke infinitely deep trees. Nevertheless, we can combine it with the results of a statistical analysis of real-world XML data [46] that the amount of repetitions of a

- I-a) $(e_1|e_2)^* \rightarrow e_1^*, e_2^*$
- I-b) $(e_1, e_2)^* \rightarrow e_1^*, e_2^*$
- I-c) $(e_1, e_2)? \rightarrow e_1?, e_2?$
- I-d) $(e_1, e_2)^+ \rightarrow e_1^+, e_2^+$
- I-e) $(e_1|e_2) \rightarrow e_1?, e_2?$

Fig. 3. Flattening rules.

- II-a) $e_1^{++} \rightarrow e_1^+$
- II-b) $e_1^{**} \rightarrow e_1^*$
- II-c) $e_1^{*?} \rightarrow e_1^*$
- II-d) $e_1^{?*} \rightarrow e_1^*$
- II-e) $e_1^{+*} \rightarrow e_1^*$
- II-f) $e_1^{*+} \rightarrow e_1^*$
- II-g) $e_1^{?+} \rightarrow e_1^*$
- II-h) $e_1^{+?} \rightarrow e_1^*$
- II-i) $e_1^{??} \rightarrow e_1?$

Fig. 4. Simplification rules.

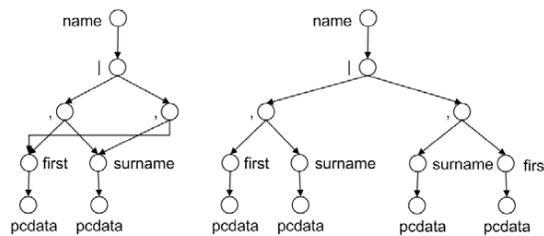


Fig. 5. Shared elements and their copies.

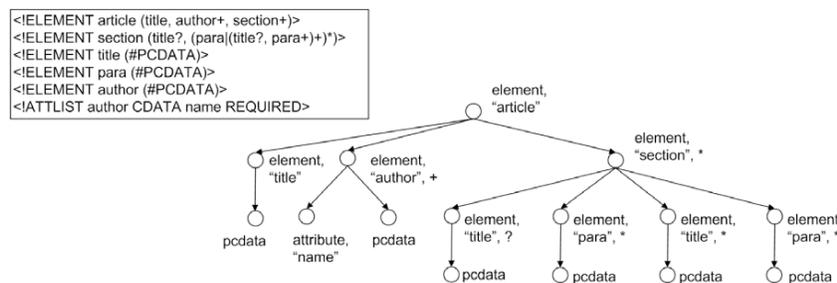


Fig. 6. An example of a DTD and its tree representation.

recursive element is in real-world XML documents very low – less than 10. Consequently, we approximate the infinite amount of repetitions with a realistic limited constant value based on observations of reality. In addition, since our approach involves operations on whole trees, it is not important exactly how many occurrences we use, because each of them can be transformed using a single edit operation.

4.1.3. DTD Tree

Having a simplified DTD without shared and recursive elements, its tree representation is defined as follows:

Definition 5. A DTD Tree is a directed ordered tree $T = (V, E)$, where

1. V is a finite set of nodes, s.t. for $\forall v \in V$, $v = (v_{Type}, v_{Name}, v_{Cardinality})$, where v_{Type} is the type of a node (i.e. attribute, element or pcdat), v_{Name} is the name of an element/attribute and $v_{Cardinality}$ is the cardinality constraint operator of an element/attribute,
2. $E \subseteq V \times V$ is a set of edges representing relationships between elements and their attributes or subelements.

An example of a DTD and its tree representation (after simplification) is depicted in Fig. 6.

4.2. Tree edit operations

Having the above described tree representation of a DTD, we can now easily utilize the tree edit algorithm proposed in [50]. Assume that we are being given a tree T with a root node r and its first-level subtrees T_1, T_2, \dots, T_m (m is denoted as a degree of tree T), then the tree edit operations are defined as follows:

Definition 6. $Substitution_T(r_{new})$ is a node substitution operation applied to T that yields the tree T' with root node r_{new} and first-level subtrees T_1, \dots, T_m .

Definition 7. Given a node x with degree 0, $Insert_T(x, i)$ is a node insertion operation applied to T at i that yields the new tree T' with root node r and first-level subtrees $T_1, \dots, T_i, x, T_{i+1}, \dots, T_m$.

Definition 8. If the first-level subtree T_i is a leaf node, $Delete_T(T_i)$ is a delete node operation applied to T at i that yields the tree T' with root node r and first-level subtrees $T_1, \dots, T_{i-1}, T_{i+1}, \dots, T_m$.

Definition 9. Given a tree T_j , $InsertTree_T(T_j, i)$ is an insert tree operation applied to T at i that yields the tree T' with root node r and first-level subtrees $T_1, \dots, T_i, T_j, T_{i+1}, \dots, T_m$.

Definition 10. $DeleteTree_T(T_i)$ is a delete tree operation applied to T at i that yields the tree T' with root node r and first-level subtrees $T_1, \dots, T_{i-1}, T_{i+1}, \dots, T_m$.

The transformation of a source tree T_x to a destination tree T_y can be done using various sequences of the operations. But, we can only deal with so-called *allowable* sequences, i.e. the relevant ones. For the purpose of our approach we only need to modify the original definition [50] as follows:

Definition 11. A sequence of edit operations transforming a source tree T_x to a destination tree T_y is *allowable* if it satisfies the following two conditions:

1. A tree T may be inserted only if a tree similar to T already occurs in the source tree T_x . A tree T may be deleted only if a tree similar to T occurs in the destination tree T_y .
2. A tree that has been inserted via the *InsertTree* operation may not subsequently have additional nodes inserted. A tree that has been deleted via the *DeleteTree* operation may not previously have had nodes deleted.

While the original definition requires exactly the same nodes and trees, the requirements can be relaxed to include only similar ones. The exact meaning of the similarity is explained in the following text and enables one to combine the tree edit distance with other similarity measures. Also note that each of the edit operations is associated with a non-negative cost as well.

4.3. Similarity of nodes of a DTD Tree

Since structural similarity is solved via the edit distance, we focus on similarity of information carried in nodes of the tree. To evaluate the similarity of the nodes u and v we exploit the semantic and syntactic similarity of element/attribute names and cardinality-constraint similarity.

Semantic similarity of element/attribute names (*SemanticSim*) is a score that reflects the semantic relation between the meanings of two words. We exploit procedure described in [34] which determines ontology similarity between two words

u_{Name} and v_{Name} by comparing u_{Name} with synonyms of v_{Name} . It exploits procedure $SynSet(w)$ which searches a thesaurus and returns the set of synonyms of a word w . At the beginning a set S is initialized as $S = \{v_{Name}\}$ and the depth of algorithm $A_{depth} = 0$. If $u_{Name} \notin S$, then $S = \bigcup_{w \in S} SynSet(w)$ and $A_{depth} = A_{depth} + 1$, until $u_{Name} \in S$ or $A_{depth} > MAX_{depth}$, where MAX_{depth} is a threshold to avoid infinite searching of the thesaurus. If no synonym is found, then $SemanticSim$ is 0, otherwise it is defined as $0.8^{A_{depth}}$.

Syntactic similarity of element/attribute names ($SyntacticSim$) is determined by computing the edit distance between u_{Name} and v_{Name} . For our purpose the classical Levenshtein algorithm [35] is used for those cases where we cannot rely on the semantics of element/attribute names, e.g. if abbreviations are used instead.

And finally, we consider similarity of the cardinality constraints ($CardSim$) of elements. This is determined by the *cardinality compatibility table* depicted in Table 1. In general, the values can be set variously – in our case we used the values proposed and verified in [34].

The overall similarity of nodes u and v is computed as:

$$Sim(u, v) = Max(SemanticSim(u, v), SyntacticSim(u, v)) \times \alpha + CardSim(u, v) \times \beta$$

where $\alpha + \beta = 1$ and $\alpha, \beta \geq 0$.

4.4. Costs of inserting and deleting trees

Inserting (deleting) a subtree T_i can be done with a single operation $InsertTree$ ($DeleteTree$) or with a combination of $InsertTree$ ($DeleteTree$) and $Insert$ ($Delete$) operations. To find the optimal variant the algorithm uses pre-computed cost for inserting T_i , $Cost_{Graft}(T_i)$ and deleting T_i , $Cost_{Prune}(T_i)$. The procedure can be divided into two parts: In the first part $ContainedIn$ list is created for each subtree of T_i ; in the second part $Cost_{Graft}$ and $Cost_{Prune}$ are computed for T_i .

4.4.1. ContainedIn lists

The procedure for determining node similarity is used for creating $ContainedIn$ lists which are then used for computing $Cost_{Graft}$ and $Cost_{Prune}$. The list is created for each node of the destination tree and contains pointers to similar nodes in the source tree (and vice versa).

The procedure for creating $ContainedIn$ lists described in Algorithm 2, lines 1–5, represents the processing of leaf nodes. In this case we simply find all nodes similar to the given node and sort them; otherwise, there is first a recursive calling of the procedure at lines 6–8, i.e. the procedure starts with leaves and continues towards the root. We then, at line 9, find all similar nodes of n in tree T_x and add them to a temporary list; after which, we have to filter the list with lists of its descendants (lines 10–12). At this step each descendant of v has to be found at a corresponding position in the descendants of nodes in the created $ContainedIn$ list. More precisely, let $u \in v_{ContainedIn}$, $children_u$ is the set of descendants of u and c is a child of v . Then $c_{ContainedIn} \cap children_u \neq \emptyset$, otherwise u is removed from $v_{ContainedIn}$. Finally, the list is again sorted (line 13).

Algorithm 2. CreateContainedInLists(T_x, v)

Input: tree T_x , root v of T_y
Output: $ContainedIn$ lists for all nodes in T_y

- 1: **if** v is leaf node **then**
- 2: $v_{ContainedIn} = FindSimilarNodes(T_x, v)$;
- 3: Sort($v_{ContainedIn}$);
- 4: return;
- 5: **end if**
- 6: **for all** child of v **do**
- 7: CreateContainedInLists($T_x, child$);
- 8: **end for**
- 9: $v_{ContainedIn} = FindSimilarNodes(T_x, v)$;
- 10: **for all** child of v
- 11: $v_{ContainedIn} = FilterLists(v_{ContainedIn}, child_{ContainedIn})$;
- 12: **end for**
- 13: Sort($v_{ContainedIn}$);

Table 1
Cardinality compatibility table.

	*	+	?	none
*	1	0.9	0.7	0.7
+	0.9	1	0.7	0.7
?	0.7	0.7	1	0.8
none	0.7	0.7	0.8	1

4.4.2. Costs of inserting trees

When the *ContainedIn* list with corresponding nodes is created for node u , the cost for inserting the tree rooted at u can be assigned. The procedure is shown in Algorithm 3. The *for* loop (lines 2–5) computes sum sum_{d_0} for inserting node u and all its subtrees. If *InsertTree* operation can be applied (*ContainedIn* list of u is not empty) sum_{d_1} is computed for this operation at line 8. The minimum of these costs is finally denoted as $Cost_{Graft}$ for node u .

Algorithm 3. ComputeCost(u)

Input: root u of T_y
Output: $Cost_{Graft}$ for T_y

- 1: $sum_{d_0} = 1$;
- 2: **for all** child of u **do**
- 3: ComputeCost(child);
- 4: $sum_{d_0} += Cost_{Graft}(child)$;
- 5: **end for**
- 6: $sum_{d_1} = \infty$;
- 7: **if** $u_{ContainedIn}$ is not empty **then**
- 8: $sum_{d_1} = ComputeInsertTreeCost(u)$;
- 9: **end if**
- 10: $Cost_{Graft}(u) = \text{Min}(sum_{d_0}, sum_{d_1})$;

4.4.3. Costs of deleting trees

Since the rules for deleting a subtree of T_x are the same as rules for inserting a subtree of T_y , costs for deleting trees are obtained by the same procedures. We only switch tree T_x with T_y in procedures *CreateContainedInLists* (Algorithm 2) and *ComputeCost* (Algorithm 3).

4.5. Computing edit distance

The last part of the algorithm, i.e. computing the edit distance, is based on dynamic programming. At this step the procedure decides which of the operations defined in Section 4.2 will be applied for each node to transform T_x to T_y . This part of algorithm does not have to be modified for DTDs, so the original procedure presented in [50] is used as depicted in Algorithm 4.

Algorithm 4. EditDistance($T_x, T_y, Cost_{Graft}$ of $T_y, Cost_{Prune}$ of T_x)

Input: tree T_x and T_y
Output: edit distance of T_x and T_y

- 1: $M = \text{degree}(T_x)$;
- 2: $N = \text{degree}(T_y)$;
- 3: $intdist[][] = \text{newint}[0..M][0..N]$;
- 4: $dist[0][0] = Cost_{Substitution}(\text{root}(T_x), \text{root}(T_y))$;
- 5: **for** $j \in [1, N]$ **do**
- 6: $dist[0][j] = dist[0][j - 1] + Cost_{Graft}(T_{y_j})$
- 7: **end for**
- 8: **for** $i \in [1, M]$ **do**
- 9: $dist[i][0] = dist[i - 1][0] + Cost_{Graft}(T_{x_i})$
- 10: **end for**
- 11: **for** $i \in [1, M]$ **do**
- 12: **for** $j \in [1, N]$ **do**
- 13: $dist[i][j] =$
 $\min(dist[i - 1][j - 1] + \text{editDistance}(T_{x_i}, T_{y_j}), dist[i][j - 1] + Cost_{Graft}(T_{y_j}), dist[i - 1][j] + Cost_{Prune}(T_{x_i}))$;
- 14: **end for**
- 15: **end for**
- 16: **return** $dist[M][N]$;

4.6. Complexity

In [50] it was proven that the overall complexity of transforming a tree T_x into a tree T_y is $O(|T_x||T_y|)$. In our method we have to consider procedures for constructing DTD Trees and for evaluating node similarity. Constructing a DTD Tree can be

done in $O(|T_x|)$ for tree T_x , so the main influence on the complexity of the algorithm have procedures *SemanticSim*, *SyntacticSim* and *CardinalitySim*. *SyntacticSim* is computed for each pair of elements in trees T_x and T_y , so the overall complexity is $O(|T_x||T_y|\omega)$, where ω is maximum length of an element/attribute label. *CardinalitySim* is also computed for each pair of elements, however, with constant complexity, i.e. in $O(|T_x||T_y|)$. The complexity of *SemanticSim* depends on the size of the thesaurus, so the overall complexity is $O(|T_x||T_y||\Sigma|)$, where Σ is the set of words in the thesaurus; it also determines the complexity of the whole algorithm.

5. Experiments

To analyze the behavior of the proposal we have created an experimental implementation of the proposed approach and then performed various tests with both synthetic and real-world XML data. The testing software was implemented in C# 2.0 [4] and the particular thesaurus used for evaluation of *SemanticSim* was *WordNet 2.1* [3]. The implementation can be downloaded from [1].

The experiments have two phases – preparatory and experimental. In the first phase we need to tune the weights and parameters of the similarity measure so that it provides realistic results. In the second phase we analyze the behavior of the proposed improvements on real-world DTDs.

5.1. Tuning of parameters

The experimental implementation enables one to set the following parameters:

- weights α and β of *Sim* (however only one of the weights needs to be set, since the value of the second one is then determined),
- node similarity threshold T_{sim} and
- cost $cost_{tree}$ of operations *InsertTree/DeleteTree* (for simplicity we assume that their costs are equal).

Most of the current papers claim that the setting of similarity parameters can be determined by a user and, hence, it is not discussed. The problem is how to prepare a reasonable setting so that the similarity measure returns reasonable results. For this purpose we use the following strategy: Firstly, we prepare a set of synthetic DTDs and we determine their mutual similarity from user's perspective. Then, we set the respective parameters so that the similarity measure returns similar results. We depict the strategy using four DTDs describing employees. Their mutual user-specified similarity is listed in Table 2.

Firstly, we set $T_{sim} = 0.5$ and $cost_{tree} = 1$ (i.e. we use generally acknowledged “reasonable” values) and analyze the results of similarity evaluation with changing weight $\alpha = 0, 0.05, 0.1, 0.15, \dots, 1$ (and respective inverse values of $\beta = 1 - \alpha$). In Fig. 7 the X-axis represents the values of α and the curves the resulting similarity of the respective pairs of DTDs. As we can see, the most reasonable values of α corresponding to similarity results expected by a user are represented using the black dots and occur within the interval of $[0.6, 0.85]$ stressed using the vertical lines. The only exception is the similarity of DTDs A and D, which demonstrates the classical situation that user-specified similarity is not precise.

Secondly, we set α and β and tune the value of T_{sim} , i.e. we perform the same set of experiments with $\alpha = 0.65$, $\beta = 0.35$, $cost_{tree} = 1$ and changing $T_{sim} = 0, 0.05, 0.1, 0.15, \dots, 1$. Their results are depicted in Fig. 8. As we can see, the optimal interval for T_{sim} , i.e. the interval where the expected similarity values (denoted again using black dots) occur is $[0.35, 0.6]$ (again stressed using the vertical lines). Note that in this case the similarity of DTDs A and D is also diverse, even beyond the scope of the results of the algorithm.

Having a reasonable setting of α , β and T_{sim} , we can now perform tests for setting $cost_{tree}$. In the current papers all the edit operations have the same cost of 1 unit; however, using the tests we want to make sure that the setting is also correct for our case. Otherwise, if we set $cost_{tree}$ too high, the operation will never be used since it will be much cheaper to use *Insert/Delete* instead. Fig. 9 depicts the results of similarity evaluation for $cost_{tree} = 0, 0.25, 0.5, 0.75, \dots, 6$. As we can see, except for the similarity of B and C the edit operations are not involved in the similarity measure if their cost is greater than 1.5, i.e. the strategy in most of the current papers is reasonable and can be used for our case too.

5.2. Experiments with real-world DTDs

According to the tuning tests in the previous experiments with synthetic data we set $\alpha = 0.65$, $\beta = 0.35$, $T_{sim} = 0.5$ and $cost_{tree} = 1$ and analyze the behavior of our similarity measure on real-world DTDs. In particular, we use the same set of 100

Table 2
Similarity of synthetic DTDs.

	A	B	C	D
A	1	0.43	0.60	0.38
B	0.43	1	0.70	0.15
C	0.60	0.70	1	0.38
D	0.38	0.15	0.38	1

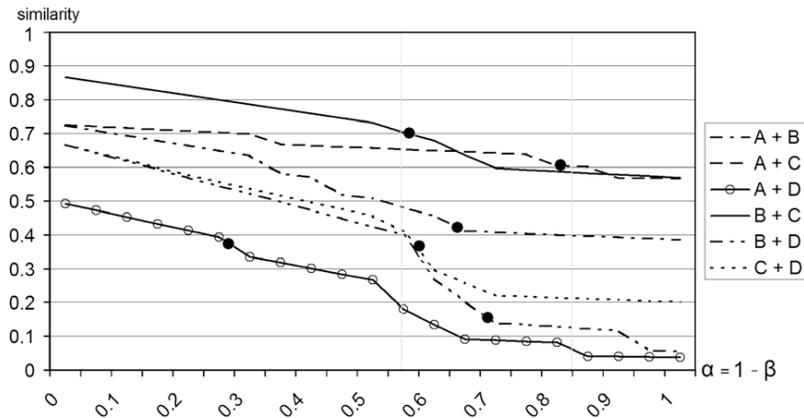


Fig. 7. Tuning of weights α (and $\beta = 1 - \alpha$) with fixed values of $T_{sim} = 0.5$ and $cost_{tree} = 1$.

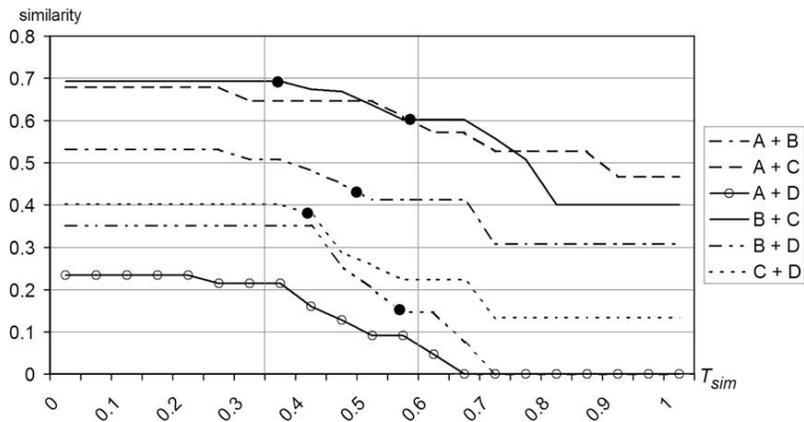


Fig. 8. Tuning of node similarity threshold T_{sim} with fixed values of with $\alpha = 0.65$, $\beta = 0.35$ and $cost_{tree} = 1$.

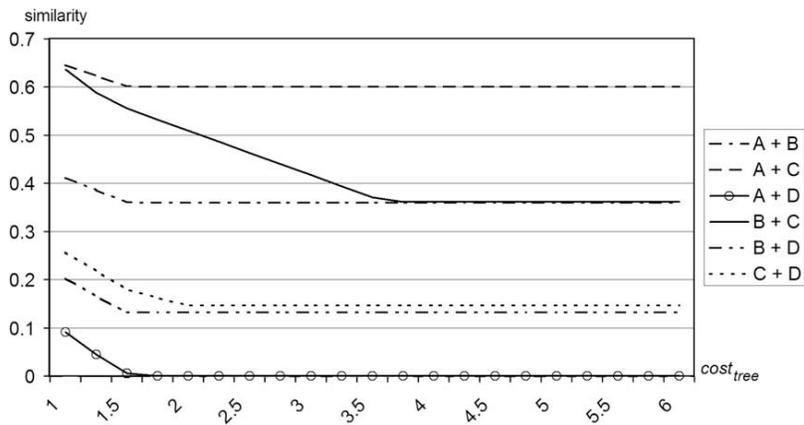


Fig. 9. Tuning of cost of *InsertTree/DeleteTree* $cost_{tree}$ with fixed values of $\alpha = 0.65$, $\beta = 0.35$, $T_{sim} = 0.5$.

XML schemas (eventually transformed to DTD) that were used in the statistical analysis of real-world XML data [46]. They can be downloaded from [2].

In the following set of test we focus on usage of all the parameters of the proposed similarity evaluation strategy, i.e. exploitation of *InsertTree/DeleteTree* operations, *CardinalitySim*, *SemanticSim* and *SyntacticSim*. As depicted in Table 3, in Test

Table 3
Setting of parameters of tests.

Test	<i>InsertTree/DeleteTree</i>	<i>CardinalitySim</i>	<i>SemanticSim</i>	<i>SyntacticSim</i>
0	×	×	×	×
1	✓	×	×	×
2	✓	✓	×	×
3	✓	×	✓	✓
4	✓	✓	✓	✓

0 we start with all the aspects disabled (denoted with ×) and then, in Tests 1–4, we gradually involve them in the similarity evaluation (denoted with ✓) and analyze the differences. Test 0 represents the basic tree edit distance and it is used only for completeness. Test 1 represents classical XML tree edit distance involving *InsertTree/DeleteTree* operations. In Test 2 we analyze the impact of *CardinalitySim*, whereas in Test 3 the impact of *SemanticSim* and *SyntacticSim*. Note that we do not analyze them separately, since, as has been mentioned, they are complementary. Finally, in Test 4 we analyze the impact of all the proposed improvements.

In all the tests we perform mutual similarity evaluation for all the 100 DTDs resulting in 4950 results. (Note that $Sim(T_x, T_y) = Sim(T_y, T_x)$.) However, due to space limitations and in the interest of clarity we show only the first 1000 most interesting results.

5.2.1. Test 0 vs. Test 1

In the first set of tests we analyze the necessity of using *InsertTree/DeleteTree* operations for the case of DTDs. For this purpose we compare Tests 0 and 1. The former one has all the parameters disabled, i.e. the similarity evaluation corresponds to classical tree edit distance strategy. In the latter one we enable usage of *InsertTree/DeleteTree* operations, i.e. it corresponds to XML tree edit distance.

The results are depicted in Fig. 10 which shows the differences between the respective results of similarity evaluation. For better clarity the results are sorted according to results of Test 0 (red curve). As we can see, there is a significant portion of cases where the results of similarity differ, i.e. adding *InsertTree/DeleteTree* operations provides more precise information. As expected, all the similarities are higher since the two more complex operations enable one to decrease the amount of simple edit operations.

In Table 4 we provide more precise information of the 4950 results of Tests 0 and 1. In particular we can see that the number of cases where the similarity changed is not high – 5.2% of results. The maximum deviation between the results of Tests 0 and 1 is 0.413, i.e. quite high. However, the average deviation is relatively low – 0.0023. It is probably caused by the fact that, in general, real-world XML schemas are relatively simple and even if they contain shared or recursive elements, they are not complex. For instance, according to the analysis [46] the most common type of recursion is the *linear* recursion which consists of a single recursive element that does not branch out.

5.2.2. Test 1 vs. Test 2

Secondly, we compare the results of Test 1, i.e. classical XML tree edit distance, with additional exploitation of similarity of cardinality constraints (*CardinalitySim*), i.e. Test 2. As depicted in Fig. 11, the differences between the similarity results are quite small (even by a magnitude smaller than in the previous case). As stated in Table 4, they are the smallest among all the performed tests. Also the number of cases that are influenced by involving this information is almost the same as in the case of the exploitation of *InsertTree/DeleteTree* operations.

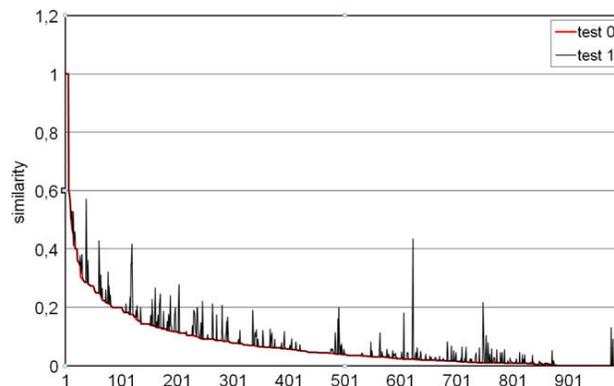
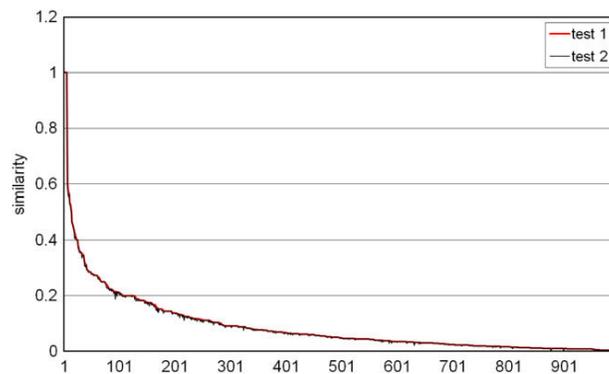
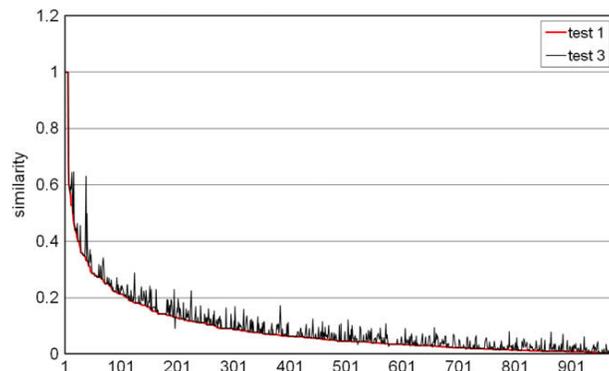


Fig. 10. Tests 0 and 1.

Table 4

Characteristics of comparison tests.

Characteristic	0 vs. 1	1 vs. 2	1 vs. 3	1 vs. 4
Percentage of changes (%)	5.2	6.8	14.2	46.7
Maximum deviation	0.4130	0.0273	0.2962	0.3314
Average deviation	0.0023	0.0002	0.0033	0.0227

**Fig. 11.** Tests 1 and 2.**Fig. 12.** Tests 1 and 3.

5.2.3. Test 1 vs. Test 3

Next, we compare the results of Test 1, i.e. classical XML tree edit distance, with the additional exploitation of similarity of element/attribute names (*SemanticSim* and *SyntacticSim*), i.e. Test 3. As we can observe in Fig. 12, in this case the deviation is quite high and so is the percentage of influenced cases (14.2%). This result is quite expected, though one may anticipate even greater differences. The problem is that it is quite common that element/attribute names do not have a proper meaning and various abbreviations or even senseless synthetic names are used instead [46]. For this purpose we have involved also *SyntacticSim*; however, it apparently cannot cope with all the cases.

5.2.4. Test 1 vs. Test 4

Finally, we compare the results of classical XML tree edit distance (Test 1) with similarity evaluation that involves all the proposed aspects, i.e. *CardinalitySim*, *SemanticSim* and *SyntacticSim* (Test 4). The results are depicted in Fig. 13 and again in Table 4. As we can see, in this case almost half of the results are modified and also the deviation of the modifications is the highest of all the tests. This confirms the assumption of all current schema matching approaches that the more partial results we include in the overall similarity measure, the more precise results we get. In this particular case we can see that the number of cases influenced is even higher than the sum of the cases influenced by the parameters separately.

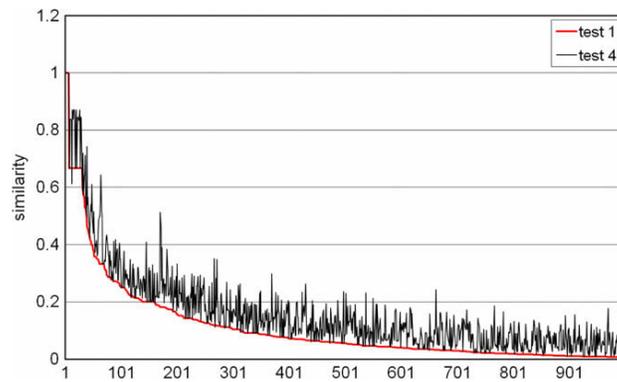


Fig. 13. Tests 1 and 4.

6. Extension to XSDs

In the previous sections we have described the proposed approach considering the DTD language. For the sake of completeness, in this section we briefly show how it can be easily extended for XSDs. As the approach has already been described in detail, we will focus only on those aspects that need to be modified. A full description of the algorithm for XSDs can be found in paper [45].

6.1. XSD constructs and their equivalence

The XML Schema language [62,11] was proposed by the W3C to address the main disadvantage of DTD – simplicity. In particular, the XML Schema language provides a wide range of simple data types (both built-in and user-defined), object-oriented features (such as inheritance or substitution), strong support for keys/foreign keys or multiple ways for reusing various parts of a schema through references. On the other hand, most of these advantages complicate the automatic processing of an XSD, because an XSD equivalent¹ to a given DTD can be written in multiple ways and the XSD itself is much more complex. An example of two XSDs equivalent to the example of a DTD in Fig. 2 are depicted in Fig. 14 examples I and II.

As we can see, the two XSDs differ in the used XML schema constructs; however, the set of XML documents valid against them is the same. In general, there are sets of XML Schema constructs that enable one to generate XSDs that have different structure but are *structurally equivalent*.

Definition 12. Let S_x and S_y be two XSD fragments. Let $I(S) = \{D \text{ s.t. } D \text{ is an XML document fragment valid against } S\}$. Then S_x and S_y are *structurally equivalent*, $S_x \sim S_y$, if $I(S_x) = I(S_y)$.

Consequently, having a set X of all XSD constructs, we can specify the quotient set X/\sim of X by \sim and respective equivalence classes – see Table 5. For instance, as depicted in Fig. 15, class C_{ST} specifies that there is no difference if a simple type is defined locally or globally. Similarly, C_{Seq} (as depicted in Fig. 16) expresses the equivalence between an unordered sequence of elements e_1, e_2, \dots, e_l and a choice of all its possible ordered permutations. Also note that each of the remaining XML Schema constructs not mentioned in Table 5 forms a single class. We will denote these classes as C_1, C_2, \dots, C_n .

Apart from XSD constructs that restrict the allowed structure of XML data, we can also find constructs that enable one to express various semantic constraints. In particular, they involve identity constraints and simple data types `ID`, `IDREF`, `IDREFS`. Hence, we can again find constructs that enable one to generate XSDs that have different structure but are *semantically equivalent*.

Definition 13. Let S_x and S_y be two XSD fragments. Then S_x and S_y are *semantically equivalent*, $S_x \approx S_y$, if they abstract the same reality.

Having a set X of all XSD constructs, we can specify the quotient set X/\approx of X by \approx and respective equivalence classes – see Table 6. Classes C'_{IdRef} and C'_{KeyRef} express the fact that both `IDREF(S)` and `keyref` constructs, i.e. foreign keys of other schema fragments, are semantically equivalent to the situation when we directly copy the referenced schema fragments to the referencing positions. An example of the equivalent schemas is depicted in Fig. 17. Similar to the previous case, each of the remaining XML Schema constructs not mentioned in Table 6 forms a single class. We will denote these classes as C'_1, C'_2, \dots, C'_m .

¹ Having the same set of instances.

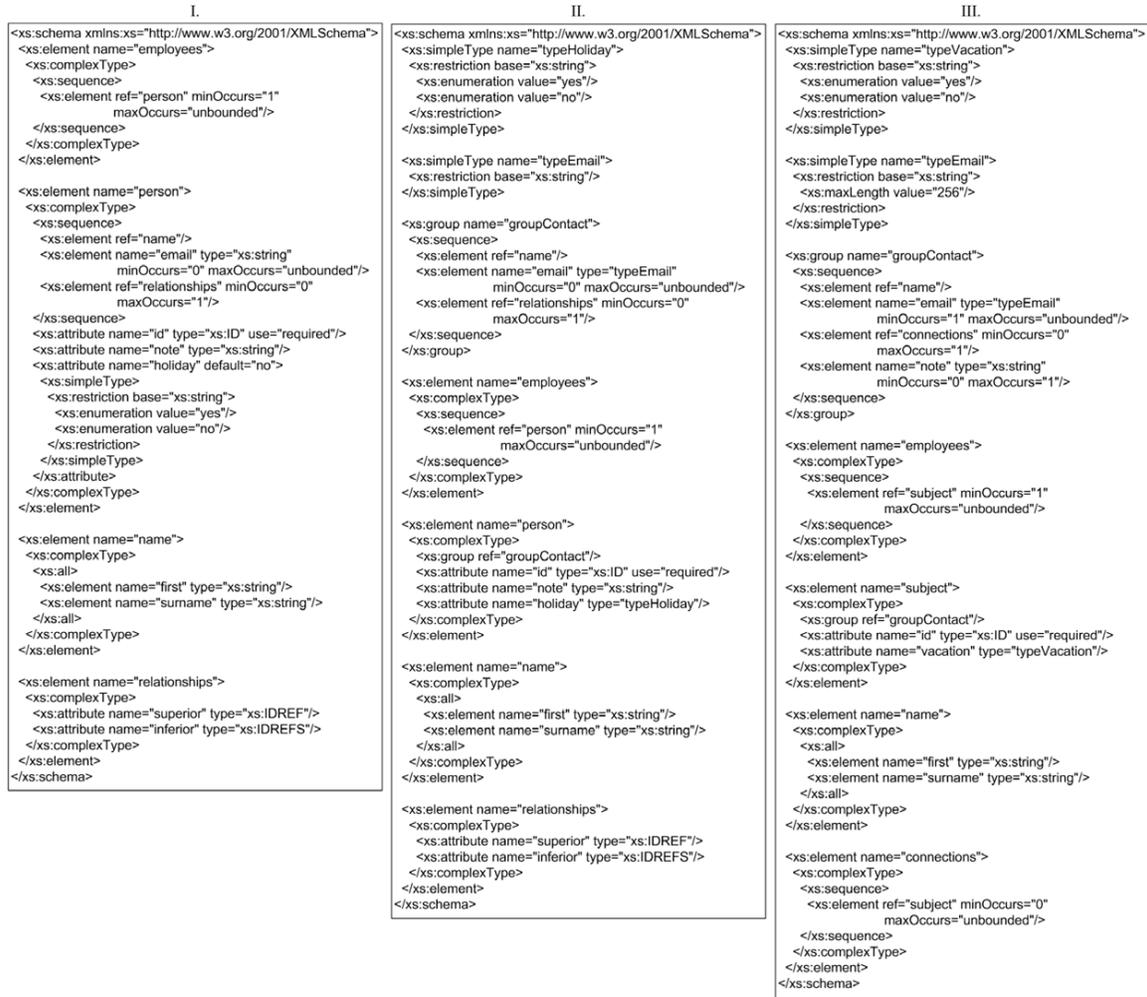


Fig. 14. Examples of XSDs.

Table 5
XSD equivalence classes of X/\sim .

Class	Constructs	Canonical representative
C_{ST}	Globally defined simple type, locally defined simple type	Locally defined simple type
C_{CT}	Globally defined complex type, locally defined complex type	Locally defined complex type
C_{El}	Referenced element, locally defined element	Locally defined element
C_{At}	Referenced attribute, locally defined attribute, attribute referenced via an attribute group	Locally defined attribute
C_{ElGr}	Content model referenced via an element group, locally defined content model	Locally defined content model
C_{Seq}	Unordered sequence of elements e_1, e_2, \dots, e_n , choice of all possible ordered sequences of e_1, e_2, \dots, e_n	Choice of all possible ordered sequences of e_1, e_2, \dots, e_n
C_{CTDer}	Derived complex type, newly defined complex type	Newly defined complex type
C_{SubSk}	Elements in a substitution group σ , choice of elements in σ	Choice of elements in σ
C_{Sub}	Data types $\tau_1, \tau_2, \dots, \tau_m$ derived from type τ , choice of content models defined in $\tau_1, \tau_2, \dots, \tau_m, \tau$	Choice of content models defined in $\tau_1, \tau_2, \dots, \tau_m, \tau$

Each of the previously defined classes of \sim or \approx equivalence can be represented using any of its elements. Since we want to simplify the specification of XSD in order to analyse its similarity, we have selected respective *canonical representatives* listed in Tables 5 and 6 as well. They enable one to simplify the structure of the XSD only to core constructs. (Note that since C_1, C_2, \dots, C_n and C'_1, C'_2, \dots, C'_m are singletons, the canonical representatives are obvious.)

<pre> ... <xs:attribute name="holiday"> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="yes"/> <xs:enumeration value="no"/> </xs:restriction> </xs:simpleType> </xs:attribute> ... </pre>	<pre> ... <xs:attribute name="holiday" type="typeHoliday"/> <xs:simpleType name="typeHoliday"> <xs:restriction base="xs:string"> <xs:enumeration value="yes"/> <xs:enumeration value="no"/> </xs:restriction> </xs:simpleType> ... </pre>
--	---

Fig. 15. Example for class C_{ST} .

<pre> ... <xs:complexType name="typeName"> <xs:all> <xs:element name="first" type="xs:string"/> <xs:element name="surname" type="xs:string"/> </xs:all> </xs:complexType> ... </pre>	<pre> ... <xs:complexType name="typeName"> <xs:choice> <xs:sequence> <xs:element name="first" type="xs:string"/> <xs:element name="surname" type="xs:string"/> </xs:sequence> <xs:sequence> <xs:element name="surname" type="xs:string"/> <xs:element name="first" type="xs:string"/> </xs:sequence> </xs:choice> </xs:complexType> ... </pre>
--	--

Fig. 16. Example for class C_{Seq} .

Table 6
XSD equivalence classes of X/\approx .

Class	Constructs	Canonical representative
C_{idRef}^1	Locally defined schema fragment, schema fragment referenced via IDREF attribute	Locally defined schema fragment
C_{keyRef}^1	Locally defined schema fragment, schema fragment referenced via keyref element	Locally defined schema fragment

6.2. Similarity of XSD constructs

Similar to the case of DTDs, the key aspect of the similarity evaluation is a tree representation of the given XSDs. We also need to simplify the content models of XSDs and to eliminate shared and repeatable elements. While the latter problem can be solved the same way as in the case of DTDs (see Section 4.1), in the former case we exploit the above defined equivalence classes.

6.2.1. Simplification of XSDs

In the first step of the simplification of XSDs we exploit structural equivalence \sim of XSD constructs and we replace each construct with the respective canonical representative. Let r be the root node of the canonical representative and, at the same time, add the respective class to the set $r_{eq.}$; the resulting schema will not contain shared schema fragments and unordered sequences.

<pre> <xs:element name="person"> <xs:complexType> <xs:sequence> <xs:element name="name" type="xs:string"/> </xs:sequence> <xs:attribute name="id" type="xs:ID"/> </xs:complexType> </xs:element> <xs:element name="relationships"> <xs:complexType> <xs:attribute name="inferior" type="xs:IDREFS"/> </xs:complexType> </xs:element> </pre>	<pre> <xs:element name="relationships"> <xs:complexType> <xs:sequence> <xs:element name="personInferior" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="name" type="xs:string"/> </xs:sequence> <xs:attribute name="id" type="xs:ID"/> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </xs:element> </pre>
--	---

Fig. 17. Example for class C_{idRef}^1 .

In the second step we exploit semantic equivalence \approx and we again replace each construct with its canonical representative r and we add the respective class to $r_{eq_{\approx}}$. Now, the resulting schema involves only elements, attributes, operators *choice* and *sequence*, intervals of allowed occurrences, simple types and assertions.

In the third step we simplify the remaining content models. For this purpose we can use the same rules as depicted in Figs. 3 and 4. Though they are expressed for DTD constructs, we can reuse them if we consider that “|” represents *choice*, “,” represents *sequence*, “?” represents interval $[0, 1]$, “+” represents intervals $[m, n]$, where $m > 0$ and $n > 1$, “*” represents intervals $[m, n]$, where $m \geq 0$ and $n > 1$ and empty operator represents interval $[1, 1]$.

6.2.2. XSD tree

Now, having a simplified XSD without shared and recursive elements, its tree representation is defined as follows:

Definition 14. An XSD tree is an ordered tree $T = (V, E)$, where

1. V is a set of nodes of the form $v = (v_{Type}, v_{Name}, v_{Cardinality}, v_{eq_{\sim}}, v_{eq_{\approx}})$, where v_{Type} is the type of a node (i.e. attribute, element or particular simple data type), v_{Name} is the name of an element or an attribute, $v_{Cardinality}$ is the interval $[m, n]$ of allowed occurrence of v , $v_{eq_{\sim}}$ is the set of classes of $\sim v$ belongs to and $v_{eq_{\approx}}$ is the set of classes of $\approx v$ belongs to,
2. $E \subseteq V \times V$ is a set of edges representing relationships between elements and their attributes or subelements.

An example of tree representation of XSD in Fig. 14 example I (after simplification) is depicted in Fig. 18.

6.2.3. Similarity evaluation

Having the tree representation of an XSD, we can now reuse most of the previously defined similarity evaluation strategy. The key difference is apparently in evaluation of similarity of XSD tree nodes u and v . In particular, we consider all the information carried in each node, i.e. semantic and syntactic similarity of element/attribute names, similarity of cardinality constraints, structural and semantic similarity of schema fragments and similarity of data types.

Semantic similarity (*SemanticSim*) and syntactic similarity (*SyntacticSim*) of element/attribute names are determined in the same way as in the case of XSDs.

On the other hand, contrary to the case of DTDs, similarity of cardinality constraints (*CardSim*) is determined by similarity of intervals $u_{Cardinality} = [u_{low}, u_{up}]$ and $v_{Cardinality} = [v_{low}, v_{up}]$. It is defined as follows:

$$\begin{aligned}
 CardSim(u, v) &= 0; && (u_{up} < v_{low}) \vee (v_{up} < u_{low}) \\
 &= 1; && u_{up}, v_{up} = \infty \wedge u_{low} = v_{low} \\
 &= 0.9; && u_{up}, v_{up} = \infty \wedge u_{low} \neq v_{low} \\
 &= 0.6; && u_{up} = \infty \vee v_{up} = \infty \\
 &= \frac{\min(u_{up}, v_{up}) - \max(u_{low}, v_{low})}{\max(u_{up}, v_{up}) - \min(u_{low}, v_{low})}; && \text{otherwise}
 \end{aligned}$$

Structural and semantic similarity of schema fragments (*StrFragSim* and *SemFragSim*) rooted at u and v is determined by the similarity of sets $u_{eq_{\sim}}, v_{eq_{\sim}}$ and $u_{eq_{\approx}}, v_{eq_{\approx}}$. It is defined as follows:

$$\begin{aligned}
 StrFragSim(u, v) &= 1; && u_{eq_{\sim}}, v_{eq_{\sim}} = \emptyset \\
 &= \frac{|u_{eq_{\sim}} \cap v_{eq_{\sim}}|}{|u_{eq_{\sim}} \cup v_{eq_{\sim}}|}; && \text{otherwise} \\
 SemFragSim(u, v) &= 1; && u_{eq_{\approx}}, v_{eq_{\approx}} = \emptyset \\
 &= \frac{|u_{eq_{\approx}} \cap v_{eq_{\approx}}|}{|u_{eq_{\approx}} \cup v_{eq_{\approx}}|}; && \text{otherwise}
 \end{aligned}$$

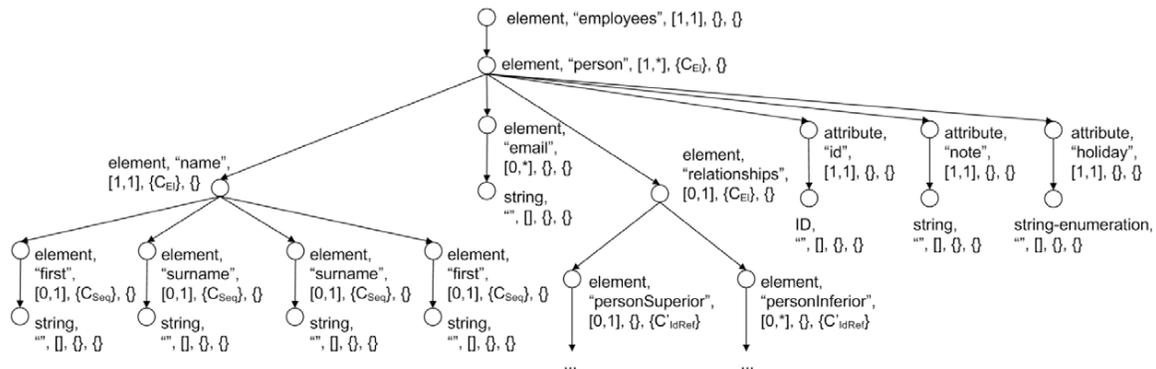


Fig. 18. An example of an XSD tree.

And, finally, similarity of data types (*DataTypeSim*) is determined by similarity of simple types u_{Type} and v_{Type} . It is specified by a *type compatibility matrix* that determines similarity in distinct simple types. For instance, similarity of `string` and `normalizedString` is 0.9, whereas similarity of `string` and `positiveInteger` is 0.5. Similarly, the table involves similarity of restrictions of simple types specified either via derivation of data types or assertions as well as similarity between element and attribute nodes. (We omit the whole table owing to its length.)

The overall similarity, $Sim(u, v)$ of nodes u and v is computed as follows:

$$Sim(u, v) = Max(SemanticSim(u, v), SyntacticSim(u, v)) \times \alpha_1 + CardSim(u, v) \times \alpha_2 \\ + StrFragSim(u, v) \times \alpha_3 + SemFragSim(u, v) \times \alpha_4 + DataTypeSim(u, v) \times \alpha_5$$

where $\sum_{i=1}^5 \alpha_i = 1$ and $\forall i: \alpha_i \geq 0$.

Complexity Since the modifications of the algorithm for the purpose of XSDs do not have higher complexity than the originals, the overall complexity is mainly determined by exploitation of *SemanticSim*, i.e. the size of the thesaurus.

6.3. Experiments

The experimental evaluation of the modifications for XSDs is similar to the DTD case. When we depict the features of the approach using the three XSDs in Fig. 14, we can see the difference between XSD I and II is only within classes of \sim equivalence. On the other hand, XSD III differs in more aspects, such as, e.g., simple types, allowed occurrences, globally/locally defined data types, exploitation of groups, element/attribute names, attributes vs. elements with simple types etc.

Table 7 depicts the results in the case we set $\alpha_3 = \alpha_4 = 0$, i.e. we ignore the information on the original constructs of XML Schema. In this case similarity of XSD I and XSD II is 1.0, because they are represented using identical XSD trees. Similarity between XSD I vs. XSD III and XSD II vs. XSD III is also equivalent for the same reasons, though naturally lower.

If we set $\alpha_3 \neq 0$ (according to our experiments it should be >0.2 to influence the algorithm), the resulting similarity is influenced by the difference between the used XML Schema constructs. The results are depicted in Table 8, where we can see more precise results. In particular, the similarity of XSD I and II is naturally $\neq 1.0$, and the similarity of XSD II and III is higher due to the respective higher structural similarity of constructs.

On the other hand, if we set $\alpha_4 \neq 0$ and $\alpha_3 = 0$, i.e. we are interested in semantic similarity of schema fragments, the results have the same trend as the results in Table 7, because we again omit the structural similarity of XSD constructs, but in this case the semantic similarity of schema fragments `relationships` and `connections` is high.

As we have mentioned, the most time consuming operation of the approach which determines the overall complexity of the algorithm is searching the thesaurus. Hence, in the last test we try to omit evaluation of *SemanticSim*. If we consider the first situation, i.e. when $\alpha_3 = \alpha_4 = 0$, it influences similarity with XSD III (which drops to 0.33), whereas similarity of XSD I and II remains the same because the respective element/attribute names are the same. The results in case $\alpha_3 \neq 0$ are depicted in Table 9. As we can see, the similarity of XSD I and II again remains the same, whereas the other values are much lower.

Table 7
Similarity for $\alpha_3 = \alpha_4 = 0$.

	XSD I	XSD II	XSD III
XSD I	1.000	1.000	0.820
XSD II	1.000	1.000	0.820
XSD III	0.820	0.820	1.000

Table 8
Similarity for $\alpha_3 \neq 0$.

	XSD I	XSD II	XSD III
XSD I	1.000	0.890	0.660
XSD II	0.890	1.000	0.700
XSD III	0.660	0.700	1.000

Table 9
Similarity without *SemanticSim*.

	XSD I	XSD II	XSD III
XSD I	1.000	0.890	0.240
XSD II	0.890	1.000	0.255
XSD III	0.240	0.255	1.000

In general, the experiments show that various parameters of the similarity measure can highly influence the results. On the other hand, we cannot simply analyze all possible aspects, since some applications may not be interested, e.g., in semantic similarity of used element/attribute names or the structurally equivalent constructs XML Schema involves.

7. Discussion

In the previous sections we have proposed an approach that enables one to evaluate similarity of XML schema fragments. Contrary to current papers (see Section 3) our approach brings several advantages and innovations.

7.1. Structural similarity

The most important advantage of our approach is a precise analysis of the structure of the compared XML schema fragments. Since most of the current approaches exploit the similarity measure for different purposes than we do, they focus mainly on semantic similarity of the given fragments, i.e. they mostly analyze the element/attribute names. In some cases the structure is taken into account – for instance leaf nodes or child nodes of the root node, paths, etc. are analyzed as well. However, a true structural analysis is not done. Since the similarity measure we propose is primarily used for the purpose of XML-to-relational mapping strategies, we focus on precise analysis of the structural aspects. The usage of edit distance enables us to fully propagate the structural differences to the resulting similarity. In addition, since we exploit more complex edit operations, i.e. inserting/deleting whole subtrees, the edit distance corresponds to features of XML data. Currently, there seems to exist only a single approach that exploits tree edit distance for XML schema similarity evaluation [29]; however, the authors deal with a special case of schemas of web service operations and, hence, they can consider only simple edit operations and schema constructs.

7.2. Exploitation of semantics

Even though we have just mentioned that our primary aim is structural analysis, we still want to incorporate semantic aspects as well. The reason being that even though the main decision for appropriate XML-to-relational storage strategy is the structure and complexity of the given XML schema fragment, the semantic aspects can provide more precise information when the given fragments are structurally highly similar, and we can still choose from multiple storage strategies.

7.3. XSD constructs

Apart from structural similarity of DTD fragments, we propose an extension of the proposed approach for XSD constructs. For this purpose we defined classes of equivalence of XSD constructs. Since we firstly preprocess the given XSDs so that they involve only canonical representatives of the equivalence classes, we are able to easily cope with the “syntactic sugar” of XML Schema language. However, since we preserve the information of the originally used constructs, we can still incorporate it to the resulting similarity if necessary. To our knowledge, this is the first approach that considers equivalency of XSD constructs in relation to similarity evaluation.

7.4. Tuning of weights

Finally, similar to current papers we exploit the idea of a weighted sum of various supplemental similarity measures. On one hand this approach requires a reasonable tuning of the weights so that the overall similarity measure returns realistic results. However, on the other hand, it enables one to easily exclude various aspects and to adapt the similarity measure to the particular application. For example, as mentioned in the previous paragraph, using the weights we can easily include or exclude the information on the originally used XSD construct.

8. Conclusion

The exploitation of similarity is a classical optimization approach for most of XML processing approaches. Hence, the area of XML similarity evaluation is wide. We can even find several papers which compare and contrast various subsets of current approaches from different points of view [20,53,57] as well as theoretic studies [58]. However, although the amount of current approaches is wide, there are still several open problems to be solved.

In this paper we focused on the problem of structural similarity of DTDs. For this purpose we have combined two approaches and adapted them to DTD-specific structure – edit distance and semantic similarity. The exploitation of edit distance enables one to analyze the structure of DTDs more precisely, whereas semantic similarity enables one to get more precise results, though at the cost of searching a thesaurus. Using a set of experiments we have demonstrated the features of the proposal. And finally, we have described its adaptation for XML Schema language and, in particular, its “syntactic sugar”.

In our future work we will focus mainly on further improvements of our approach. The first possible improvement can be found in using other edit operations such as, e.g. moving a node or adding/deleting a non-leaf node. Secondly, simplification rules which are satisfactory for our primary purpose should be improved, because the loss of information is relatively high in general. Another remaining issue that persists in most similarity approaches that involve a kind of weighted sum is the question of reasonable setting of the weights. In general, all such approaches consider the possibility of setting the weights according to user's requirement as an advantage. However, the problem is how to find such reasonable setting.

The key aim of our future work will be to find further ways of exploiting our proposal, i.e. areas that can be optimized with its usage. The possible range is relatively wide and involves problems such as XML query evaluation [7,33,43], XML data storage strategies [67,44], XSLT transformations [24,27], reverse engineering of XML data [48] or even searching of best K objects [51,36]. However, each such strategy has different requirements for the similarity evaluation and, hence, it needs to be appropriately adapted.

Acknowledgement

We would sincerely like to thank all our reviewers for their careful reading of the paper and for their extremely valuable comments and suggestions. Our thanks also go to our sponsors, in particular this work was supported in part by the Czech Science Foundation (GAČR), Grant No. 201/09/P364.

References

- [1] <http://www.ksi.mff.cuni.cz/mlynkova/xmlsim/app.zip>.
- [2] <http://www.ksi.mff.cuni.cz/mlynkova/xmlsim/dtd.zip>.
- [3] WordNet – Lexical Database for the English Language, Princeton University, 2006. <<http://wordnet.princeton.edu/>>.
- [4] The C# Language, Microsoft Corporation, 2008. <<http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx>>.
- [5] A. Algergawy, E. Schallehn, G. Saake, Improving XML schema matching performance using pruffer sequences, *Data Knowl. Eng.* 68 (8) (2009) 728–747.
- [6] M. Altinel, M.J. Franklin, Efficient filtering of XML documents for selective dissemination of information, in: VLDB'00: Proceedings of 26th International Conference on Very Large Data Bases, Morgan Kaufmann, San Francisco, CA, USA, 2000, pp. 53–64.
- [7] D. Bednarek, Output-driven XQuery evaluation, in: *Intelligent Distributed Computing, Systems and Applications, Studies in Computational Intelligence*, vol. 162, Springer-Verlag, 2008, pp. 55–64.
- [8] J. Berstel, L. Boasson, XML grammars, in: *Mathematical Foundations of Computer Science, LNCS*, Springer, 2000, pp. 182–191.
- [9] E. Bertino, G. Guerrini, M. Mesiti, A matching algorithm for measuring the structural similarity between an XML document and a DTD and its applications, *Inform. Syst.* 29 (1) (2004) 23–46.
- [10] E. Bertino, G. Guerrini, M. Mesiti, I. Rivara, C. Tavella, Measuring the Structural Similarity among XML Documents and DTDs, Technical Report DISI-TR-02-02, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 2001. <<ftp://ftp.disi.unige.it/person/MesitiM/papers/BGM-xml.pdf>>.
- [11] P.V. Biron, A. Malhotra, XML schema part 2: datatypes (second ed.), W3C, 2004. <www.w3.org/TR/xmlschema-2/>.
- [12] A. Bonifati, G. Mecca, A. Pappalardo, S. Raunich, G. Summa, Schema mapping verification: the spicy way, in: EDBT'08: Proceedings of the 11th International Conference on Extending Database Technology, ACM, New York, NY, USA, 2008, pp. 85–96.
- [13] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, F. Yergeau, Extensible Markup Language (XML) 1.0 (fourth ed.), W3C, 2006. <<http://www.w3.org/TR/REC-xml/>>.
- [14] E.R. Canfield, G. Xing, Approximate XML document matching, in: SAC'05: Proceedings of the 2005 ACM Symposium on Applied Computing, ACM Press, New York, NY, USA, 2005, pp. 787–788.
- [15] S.S. Chawathe, Comparing hierarchical data in external memory, in: VLDB'99: Proceedings of the 25th International Conference on Very Large Data Bases, Morgan Kaufmann, San Francisco, CA, USA, 1999, pp. 90–101.
- [16] S.S. Chawathe, H. Garcia-Molina, Meaningful change detection in structured data, in: SIGMOD'97: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, ACM Press, New York, NY, USA, 1997, pp. 26–37.
- [17] I. Choi, B. Moon, H.-J. Kim, A clustering method based on path similarities of XML data, *Data Knowl. Eng.* 60 (2) (2007) 361–376.
- [18] G. Cobena, S. Abiteboul, A. Marian, Detecting changes in XML documents, in: ICDE'08: Proceedings of the 18th International Conference on Data Engineering, IEEE Computer Society, San Jose, CA, USA, 2002, pp. 41–52.
- [19] T. Dalamagas, T. Cheng, K.-J. Winkel, T. Sellis, A methodology for clustering XML documents by structure, *Inform. Syst.* 31 (3) (2006) 187–228.
- [20] H. Do, S. Melnik, E. Rahm, Comparison of schema matching evaluations, in: Revised Papers from the NODe'02 Web and Database-Related Workshops on Web, Web-Services, and Database Systems, Springer-Verlag, London, UK, 2003, pp. 221–237.
- [21] H.H. Do, E. Rahm, COMA – a system for flexible combination of schema matching approaches, in: VLDB'02: Proceedings of the 28th International Conference on Very Large Data Bases, Morgan Kaufmann, Hong Kong, China, 2002, pp. 610–621.
- [22] H.-H. Do, E. Rahm, Matching large schemas: approaches and evaluation, *Inform. Syst.* 32 (6) (2007) 857–885.
- [23] F. Duchateau, Z. Bellahsene, R. Coletta, A flexible approach for planning schema matching algorithms, in: OTM'08: Proceedings of the OTM 2008 Federated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 249–264.
- [24] J. Dvorakova, F. Zavoral, Using Input Buffers for Streaming XSLT Processing, in: Proceedings of the First International Conference on Advances in Databases, Knowledge, and Data Applications, IEEE Computer Society, Los Alamitos, CA, USA, 2009, pp. 50–55.
- [25] H. Elmeleegy, M. Ouzzani, A. Elmagarmid, Usage-based schema matching, in: ICDE'08: International Conference on Data Engineering, 2008, pp. 20–29.
- [26] S. Flesca, G. Manco, E. Masciari, L. Pontieri, A. Pugliese, Detecting structural similarities between XML documents, in: WebDB'02: Proceedings of the Fifth International Workshop on the Web and Databases, Madison, Wisconsin, USA, 2002, pp. 55–60.
- [27] S. Groppe, J. Groppe, Output schemas of XSLT stylesheets and their applications, *Inform. Sci.* 178 (21) (2008) 3989–4018.
- [28] R.W. Hamming, Error detecting and error correcting codes, *Bell Syst. Tech. J.* 26 (2) (1950) 147–160.
- [29] Y. Hao, Y. Zhang, Web services discovery based on schema matching, in: ACSC'07: Proceedings of the 30th Australasian Conference on Computer Science, Australian Computer Society, Inc., Darlinghurst, Australia, 2007, pp. 107–113.
- [30] B. He, K.C.-C. Chang, A holistic paradigm for large scale schema matching, *SIGMOD Rec.* 33 (4) (2004) 20–25.
- [31] B. Jeong, D. Lee, H. Cho, J. Lee, A novel method for measuring semantic similarity for XML schema matching, *Expert Syst. Appl.* 34 (3) (2008) 1651–1658.
- [32] T. Jiang, L. Wang, K. Zhang, Alignment of trees – an alternative to tree edit, *Theor. Comput. Sci.* 143 (1) (1995) 137–148.
- [33] H.-H. Lee, W.-S. Lee, Selectivity-sensitive shared evaluation of multiple continuous XPath queries over XML streams, *Inform. Sci.* 179 (12) (2009) 1984–2001.

- [34] M.L. Lee, L.H. Yang, W. Hsu, X. Yang, XClust: clustering XML schemas for effective integration, in: CIKM'02: Proceedings of the 11th International Conference on Information and Knowledge Management, ACM Press, New York, NY, USA, 2002, pp. 292–299.
- [35] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals, *Sov. Phys. Dokl.* 10 (1966) 707.
- [36] G. Li, C. Li, J. Feng, L. Zhou, SAIL: structure-aware indexing for effective and progressive top-K keyword search over XML documents, *Inform. Sci.* 179 (21) (2009) 3745–3762.
- [37] J. Madhavan, P.A. Bernstein, A. Doan, A. Halevy, Corpus-based schema matching, in: ICDE'05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05), IEEE Computer Society, Washington, DC, USA, 2005, pp. 57–68.
- [38] J. Madhavan, P.A. Bernstein, E. Rahm, Generic schema matching with cupid, in: VLDB'01: Proceedings of the 27th International Conference on Very Large Data Bases, Morgan Kaufmann, San Francisco, CA, USA, 2001, pp. 49–58.
- [39] S. Melnik, H. Garcia-Molina, E. Rahm, Similarity flooding: a versatile graph matching algorithm and its application to schema matching, in: ICDE'02: Proceedings of the 18th International Conference on Data Engineering, IEEE Computer Society, Washington, DC, USA, 2002, p. 117.
- [40] P.D. Meo, G. Quattrone, G. Terracina, D. Ursino, Integration of XML schemas at various “severity” levels, *Inform. Syst.* 31 (6) (2006) 397–434.
- [41] F. Meziane, Y. Rezgui, A document management methodology based on similarity contents, *Inform. Sci.* 158 (1) (2004) 15–36.
- [42] T. Milo, S. Zohar, Using schema matching to simplify heterogeneous data translation, in: VLDB'98: Proceedings of 24th International Conference on Very Large Data Bases, Morgan Kaufmann, San Francisco, CA, USA, 1998, pp. 122–133.
- [43] J.-K. Min, M.-j. Park, C.-W. Chung, XTREAM: an efficient multi-query evaluation on streaming XML data, *Inform. Sci.* 177 (17) (2007) 3519–3538.
- [44] I. Mlynkova, A journey towards more efficient processing of XML data in (O)RDBMS, in: CIT'07: Proceedings of the Seventh International Conference on Computer and Information Technology, IEEE Computer Society, Fukushima, Japan, 2007, pp. 23–28.
- [45] I. Mlynkova, Equivalence of XSD constructs and its exploitation in similarity evaluation, in: ODBASE'08, LNCS, vol. 5332, Springer-Verlag, Monterrey, Mexico, 2008, pp. 1253–1270.
- [46] I. Mlynkova, K. Toman, J. Pokorny, Statistical analysis of real XML data collections, in: COMAD'06: Proceedings of the 13th International Conference on Management of Data, Tata McGraw-Hill Publishing, New Delhi, India, 2006, pp. 20–31.
- [47] R. Nayak, Fast and effective clustering of XML data using structural information, *Knowl. Inform. Syst.* 14 (2) (2008) 197–215.
- [48] M. Necasky, Reverse engineering of XML schemas to conceptual diagrams, in: APCCM'09: Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modelling (to appear), CRPIT, Australian Computer Society, Inc., 2009.
- [49] P.K. Ng, V.T. Ng, Structural similarity between XML Documents and DTDs, in: ICCS'03: Proceedings of the Third International Conference on Computational Science, Springer-Verlag, 2003, pp. 412–421.
- [50] A. Nierman, H.V. Jagadish, Evaluating structural similarity in XML documents, in: WebDB'02: Proceedings of the Fifth International Workshop on the Web and Databases, Madison, Wisconsin, USA, 2002, pp. 61–66.
- [51] M. Ondreicka, J. Pokorny, Extending Fagin's algorithm for more users based on multidimensional B-tree, in: Advances in Databases and Information Systems, Lecture Notes in Computer Science, vol. 5207, Springer, 2008, pp. 199–214.
- [52] D. Rafiei, D.L. Moise, D. Sun, Finding syntactic similarities between XML documents, in: DEXA'06: Proceedings of the 17th International Conference on Database and Expert Systems Applications, IEEE Computer Society, Washington, DC, USA, 2006, pp. 512–516.
- [53] E. Rahm, P.A. Bernstein, A survey of approaches to automatic schema matching, *VLDB J.* 10 (4) (2001) 334–350.
- [54] K. Saleem, Z. Bellahsene, E. Hunt, PORSCHE: Performance ORiented SCHEMA Mediation, *Inform. Syst.* 33 (7–8) (2008) 637–657.
- [55] S. Sellami, A.-N. Benharkat, R. Rifaieh, Y. Amghar, Extension of schema matching platform ASMADE to constraints and mapping expression, 2009, pp. 223–234.
- [56] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. DeWitt, J.F. Naughton, Relational databases for querying XML documents: limitations and opportunities, in: VLDB'99: Proceedings of 25th International Conference on Very Large Data Bases, Morgan Kaufmann, San Francisco, CA, USA, 1999, pp. 302–314.
- [57] P. Shvaiko, J. Euzenat, A survey of schema-based matching approaches, *J. Data Semantics IV* 3730 (2005) 146–171.
- [58] M. Smiljanic, M. van Keulen, W. Jonker, Formalizing the XML schema matching problem as a constraint optimization problem, in: DEXA'05: Proceedings of the 20th International Conference on Database and Expert Systems Applications, LNCS, vol. 3588, Springer-Verlag, 2005, pp. 333–342.
- [59] M. Smiljanic, M. van Keulen, W. Jonker, Using element clustering to increase the efficiency of XML schema matching, in: ICDEW'06: Proceedings of the 22nd International Conference on Data Engineering Workshops, IEEE Computer Society, Los Alamitos, CA, USA, 2006, pp. 45–54.
- [60] K.-C. Tai, The tree-to-tree correction problem, *J. ACM* 26 (3) (1979) 422–433.
- [61] N. Tansalarak, K.T. Claypool, QMatch – using paths to match XML schemas, *Data Knowl. Eng.* 60 (2) (2007) 260–282.
- [62] H.S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, XML Schema part 1: structures (second ed.), W3C, 2004. <www.w3.org/TR/xmlschema-1/>.
- [63] K. Thompson, Programming techniques: regular expression search algorithm, *Commun. ACM* 11 (6) (1968) 419–422.
- [64] X. Wan, A novel document similarity measure based on earth mover's distance, *Inform. Sci.* 177 (18) (2007) 3718–3730.
- [65] L. Wang, K. Zhang, K. Jeong, D. Shasha, A system for approximate tree matching, *IEEE Trans. Knowl. Data Eng.* 6 (4) (1994) 559–571.
- [66] A. Wojnar, I. Mlynkova, J. Dokulil, Similarity of DTDs based on edit distance and semantics, IDC'08, Studies in Computational Intelligence, vol. 162, Springer-Verlag, Catania, Italy, 2008, pp. 207–216.
- [67] J. Yaghob, F. Zavoral, Semantic Web Infrastructure using DataPile, in: The 2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, IEEE Press, Los Alamitos, California, 2006, pp. 630–633.
- [68] T.W. Yan, H. Garcia-Molina, The SIFT information dissemination system, *ACM Trans. Database Syst.* 24 (4) (1999) 529–565.
- [69] S. Yi, B. Huang, W.T. Chan, XML application schema matching using similarity measure and relaxation labeling, *Inform. Sci.* 169 (1–2) (2005) 27–46.
- [70] A. Zerdazi, M. Lamolle, Computing path similarity relevant to XML schema matching, in: OTM'08: Proceedings of the OTM Confederated International Workshops and Posters on On the Move to Meaningful Internet Systems, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 66–75.
- [71] Z. Zhang, R. Li, S. Cao, Y. Zhu, Similarity metric for XML documents, in: FGWM'03: Proceedings of the International Workshop on Knowledge and Experience Management, Karlsruhe, Germany, 2003.

Chapter 6

Analyzer: A Complex System for Data Analysis

Jakub Stárka
Martin Svoboda
Jan Sochna
Jiří Schejbal
Irena Mlýnková
David Bednárek

Published in the *Computer Journal*, volume 55, issue 5, pages 590–615. Oxford University Press, 2012. ISSN 0010-4620.

Impact Factor: 0.785
5-Year Impact Factor: 0.943



Analyzer: A Complex System for Data Analysis

JAKUB STÁRKA*, MARTIN SVOBODA, JAN SOCHNA, JIŘÍ SCHEJBAL,
IRENA MLÝNKOVÁ AND DAVID BEDNÁREK

*Department of Software Engineering, Charles University in Prague, Malostranské nám. 25, 118 00 Praha 1,
Czech Republic*

**Corresponding author: starka@ksi.mff.cuni.cz*

Recently eXtensible Markup Language (XML) has achieved the leading role among languages for data representation and, thus, we can witness a massive boom of corresponding techniques for managing XML data. Most of the processing techniques, however, suffer from various bottlenecks worsening their time and/or space efficiency. We assume that the main reason is they consider XML collections too globally, involving all their possible features, although real-world data are often much simpler. Even though some techniques do restrict the input data, the restrictions are mostly unnatural. This paper aims to introduce *Analyzer*—a complex framework for performing statistical analyses of real-world documents. Exploitation of results of these analyses is a classical way how data processing can be optimized in many areas. Although this intent is legitimate, *ad hoc* and dedicated analyses soon become obsolete, they are usually built on insufficiently extensive collections and are difficult to repeat. *Analyzer* represents an easily extensible framework, which helps the user with gathering documents, managing analyses and browsing computed reports.

*Keywords: XML data analysis; XML data crawling; XML data correction; structural analysis;
XQuery analysis*

*Received 9 May 2011; revised 7 August 2011
Handling editor: Ethem Alpaydin*

1. INTRODUCTION

The eXtensible Markup Language (XML) [1] is currently a *de facto* standard for data representation. Its popularity is given by the fact that it is well defined, easy to use and, at the same time, powerful enough. Firstly, XML was only exploited as a syntax for parametrization files. However, with the growing popularity of advanced XML technologies (such as XML Schema [2, 3], XPath [4], XQuery [5], XSLT [6] etc.) there appeared also true XML applications that exploit the whole family of XML standards for managing, processing, exchanging, querying, updating and compressing XML data that mutually compete in speed, efficiency and minimum space and/or memory requirements. Similarly, there occurred a huge amount of standards based on XML technologies, such as WSDL [7], SVG [8], RDF [9], OpenOffice [10] etc., that exploit the advantages of XML for specific purposes. Unfortunately, for a majority of these techniques and applications there can be found a number of drawbacks concerning their efficiency.

Under closer investigation we can distinguish two situations. On the one hand, there is a group of general techniques that take into account all possible features of input XML data. This idea is obviously correct, but the problem is that the XML standards were proposed in full possible generality so that future users can choose what suits them most. Nevertheless, the real-world XML data are usually not so ‘rich’, thus the effort spent on every possible feature is mostly useless. From the point of view of structural or space efficiency, it can even be harmful. On the other hand, there are techniques that somehow do restrict features of the given input XML data. For them it is natural to expect inefficiencies to occur only when the given data do not correspond to these restrictions. (In extreme cases, selected approaches even do not support any other data than those that they can process efficiently.) The problem is that such restrictions do not result from features of real-world XML applications and requirements, but they are often caused by limitations of a particular technique, complexity of such a

solution, irregularities etc. Consequently, the restrictions are not natural and do not reflect user requirements.

We can naturally pose two apparent questions:

- (i) Is it necessary to take into account a feature that will be used minimally or will not be used at all?
- (ii) If so, what are these features?

The answer for the first question obviously depends on the particular situation, i.e. application. The second question can be answered only using a detailed analysis of a sample set of real-world XML documents. However, working with real-world data is not simple, since they can often change, are not precise or even involve a number of errors. In our approach, we have addressed the following four problems:

- (i) *Data crawling*: there exists a huge number of web crawlers; however, their filters and crawling strategies must be retargeted from HTML to the XML family of documents.
- (ii) *Processing of incorrect data*: since the data are usually human-written, they contain a number of errors. In this case we can either discard the incorrect data and, hence, lose a significant portion of them, or provide a kind of *corrector*.
- (iii) *Structural analysis*: a number of features like size, depth or fan-out may be statistically examined in a collection of XML documents; similar analysis may also be applied to XML schemas in any form.
- (iv) *Query analysis*: a collection of XQuery, XSLT or XPath queries may be examined for the presence of certain query language constructs or their combinations. Because of the complexity of the query languages and the large variety of motivations for such examination, the analytical tool must be as generic as possible.

In addition, we have to cope with the fact that the data can change and, hence, the analytical phase must be repeatable and extensible. And, finally, having obtained the results of the statistics, we need to be able to visualize and analyze the huge amount of information efficiently and mutually compare the results.

In this paper, we describe a proposal background, architecture outline, implementation aspects and usage scenarios of a general framework called *Analyzer* that aims to cope with all the previously named requirements. In other words, it provides all the essential functionality for an easy management of files to be analyzed, configuration and execution of selected analyses and an advanced graphical user interface (GUI) for browsing generated reports. The key advantage of *Analyzer* is extensibility. This not only means the ability to implement own and more suitable kernel components responsible, e.g. for storing computed analytical data, but primarily the open concept of plugins. *Analyzer* provides a general environment, whereas all analytical computations themselves are defined solely within the implementation of plugins. The user is therefore expected to

first install *Analyzer* itself and then create his/her own plugins designed to correspond to the determined research intents. Although our initial motivations were related to XML data, *Analyzer* usage is not limited only to this area.

Contributions. The key contributions of this paper can be summed up as follows:

- (i) We introduce the architecture and functionality of *Analyzer*. To our knowledge it is a unique application that enables to perform automatic, repeatable and extensible analyses of real-world data. It currently supports modules for XML data processing and analyses, however, using plugins it can be extended to any kind of data.
- (ii) *Analyzer* is a complex tool that supports not only the analytical part, but also various ‘supportive’ functions, such as data crawling or data correction, as well as user-related features, such as definition of projects, visualization of results etc.
- (iii) With regard to the current support of XML, we study and describe four related issues—XML data crawling, XML data correction, structural analysis of XML data and query analysis of XML queries. In the former three cases we provide significant extensions to the current approaches; in the latter case we provide a unique approach which has not been considered in the current papers so far.
- (iv) We provide an overview of the current-related work, in particular, results of statistical analyses of real-world XML data. It enables us to show that all the results can easily be covered by *Analyzer*, further extended and repeated so that data changes and application evolution can be studied. Again, to our knowledge, such a feature has not been considered in the recent literature so far.

Outline. The paper is structured as follows. In Section 2 we provide a brief motivating discussion which outlines the necessity of analyses of real-world XML data. In Section 3 we describe the architecture of *Analyzer*, which indicates its general functionality. The next four sections are devoted to four key parts of *Analyzer*—data crawling (Section 4), processing of incorrect data (Section 5), structural analysis (Section 6) and query analysis (Section 7). In Section 8 we describe the related papers relevant to our research and in Section 9 we indicate remaining open problems and possible future extensions of *Analyzer*. Finally, in Section 10 we conclude.

Relation to previous work. In this paper, we partially exploit, combine and, in particular, extend our several previous results. Motivated by a successful and interesting statistical analysis of real-world XML data [11], *Analyzer* was implemented as an SW project of Master students of the Department of Software

Engineering of the Charles University in Prague. Its installation package as well as documentation and source files can be found at its official website [12]. Its first release 1.0 involved only basic functionality to demonstrate its key features and advantages and it was briefly introduced in paper [13]. Its four key parts were then extended in four master theses of its authors supervised by Irena Mlýnková and David Bednárek. In particular, Jan Sochna [14] focussed on the primary aspect of *Analyzer*—efficient crawling of XML data. Svoboda [15] proposed several improvements of algorithms for correction of XML data. Stárka [16] focussed on an important aspect of the structural analysis of XML data, i.e. XML similarity. And, finally, Schejbal [17] dealt in his thesis in current a most open topic of analysis of XML operations, i.e. XQuery queries. In the following text we describe and put into context the key results of the four theses and, in particular, show their close connection and resulting advantages and contributions they bring. Our aim is to provide a thorough description of all aspects of *Analyzer* as well as data analysis in general, useful for both future users of *Analyzer* and researchers dealing with related issues.

2. MOTIVATION

The idea of exploitation of the knowledge of real-world data is not new and currently we can find several applications and use cases, where it is successfully applied. From the general point of view we are interested in the subset of constructs and structures allowed by a particular language that is commonly used in practice. In this section, we provide several examples, where the knowledge of real-world data has been successfully exploited.

Incorrect assumptions on real-world data. One of the most important advantages of statistical analyses of real-world data are refutation of incorrect assumptions on typical use cases, features of the data, their complexity etc. As an example we can consider two distinct cases—schema-driven XML-to-relational storage strategies and exploitation of recursion.

Schema-driven XML-to-relational mapping methods [18, 19] are based on an existing schema S_1 of stored XML documents which is mapped to an (object-)relational database schema S_2 . The data from XML documents valid against S_1 are then stored into relations of S_2 . The purpose of these methods is to create an optimal schema S_2 , which consists of a reasonable amount of relations and whose structure corresponds to the structure of S_1 as much as possible. Naturally, such approaches require a presence of an XML schema. However, statistical analyses of real-world XML data show that a significant portion of XML documents (52% [20] of randomly crawled or 7.4% [11] of semi-automatically collected) still have no schema at all. What is more, XML Schema definitions (XSDs) are used even less (only for 0.09% [20] of randomly crawled or 38% [11] of semi-automatically collected XML documents) and even if they are

used, they often (in 85% of cases [21]) define so-called *local tree grammars* [22], i.e. grammars that can be defined using DTD as well. Hence, these methods need to be accompanied with approaches for inference of an XML schema for a given set of XML documents [23, 24].

Conversely, the support for recursion is often neglected and it is considered as a side/auxiliary construct. However, analyses show that in selected types of XML data it is used quite often (in 58% of all DTDs [25], or in 43% of document-centric and 64% of exchange documents [11]) and, hence, its efficient support is very important. On the other hand, the number of distinct recursive elements is typically low (for each category less than 5) and that the type of recursion commonly used is very simple.

Efficient processing of XML data with limited complexity. Another important observation common to all the current papers describing results of statistical analyses of real-world data is that the data is usually much simpler than the respective standard allows. Such a feature opens a wide range of optimization strategies that do not have to consider the full generality of the standard, but can count on some limitations. One of the most surprising observations of this kind is that the average depth of XML documents is < 10 , mostly around 5. This information is already widely exploited in techniques [26, 27] that represent XML documents as a set of points in multidimensional space and store them in corresponding data structures, e.g. *R-trees*, *UB-trees* [28] or *BUB-trees* [29], for the purpose of efficient querying. The dimension of the space is given by the depth of the data, and so the lower, the better.

Restriction to real-world data structures. A situation similar to the previous one occurs in cases when we can restrict a particular approach only to cases that occur in real-world data and, hence, have a reasonable basis. A research area that widely exploits the knowledge of complexity of real-world data is inference of XML schemas from a given sample set of XML documents. Since according to Gold's theorem [30] regular languages (i.e. those generated by XML schemas) are not identifiable only from positive examples (i.e. sample XML documents which should be valid against the resulting schema), the existing methods need to exploit either heuristics or a restriction to an *identifiable* subclass of regular languages. The question is which subclass should be considered. The authors of papers [31, 32] obtain a result from their analysis of real-world XML schemas [21] and define the classes so that they cover most of the real-world examples. So the identifiable subclass corresponds to a realistic situation.

Tuning of weights and parameters. Last but not least typical example of exploitation of characteristics of real-world XML data can be found in reasonable and realistic setting of various weights, parameters and characteristics of different approaches. As an example we can consider two XML

applications—evaluation of similarity of XML schemas and adaptive XML-to-relational storage strategies.

Similarity of XML schemas is currently exploited in many approaches, typically as a kind of optimization heuristics. The current approaches [33–35] are based on the idea of exploitation of various *matchers*, i.e. functions that evaluate similarity of selected simple data characteristics (e.g. similarity of element names, similarity of number of subelements etc.). Their results are then aggregated to a resulting *composite similarity measure* using a kind of weighted sum. The problem is how to set the weights so that the result reflects the reality. And the solution can be found in statistical analyses of real-world data.

In case of adaptive XML-to-relational storage strategies we need to solve a similar problem. The approaches [36, 37] focus on the idea that each application requires a different storage strategy to achieve optimal efficiency. So, before they provide the resulting mapping, they analyze a given set of sample data and operations which represent the target application and adapt the resulting mapping accordingly. Hence, again, an analysis of real-world data is crucial so that the algorithm works correctly and efficiently.

Apparently, in all the described examples we need to know the structure and complexity of the real-world data as precisely as possible. What is more, since user requirements often change and new applications occur every day, we also need to know whether and how the data characteristics evolve and adapt the approaches and optimizations accordingly.

3. FRAMEWORK ARCHITECTURE

This section is concerned with the architecture of *Analyzer*, the proposed analytical model and basic implementation aspects.

3.1. Framework architecture

The implementation of *Analyzer* allows one to work with multiple opened *projects* at once, each representing one analytical research intent. Thus, we can divide the framework architecture into two separate levels, as it is depicted in Fig. 1. The first one contains components, which are shared by all the projects. The second one represents components exclusively used and created in each opened project separately.

Project components. Components at the project level involve particularly *repositories*, *storages* and *crawlers*. They are exclusively owned by each project, but this does not mean that, e.g. a real relational database server behind a repository cannot be used by multiple projects. This is allowed and a given component only has to ensure the required isolation between individual projects (which can be done easily in the given example using different databases).

First, each project must have a single *repository*. It serves for storing all computed analytical data and the majority of project configuration metadata. Although the design of *Analyzer* does not require it, all provided repository implementations are based on standard relational databases. Secondly, *storages* are used for storing document contents, i.e. binary contents of analyzed files. The only stable implementation is based on a native file system, but experiments were taken with native storages for XML documents, too. Thirdly, there are two ways how we can insert files to be analyzed into a created project. First, we are able to import them from a specified storage, or we can use *crawlers* to download them from the Web. The download process may involve accessing of explicitly required files, or the crawler itself may be able to attempt to find other referenced files using link traversal, limited only by a maximum allowed searching depth.

An essential design feature of all these three components is extensibility. Although a typical user would probably not need it, new components can be implemented and added relatively easily into the entire system.

The project layer also contains a set of *managers*, which are responsible for creating, editing and processing of all entities such as *documents*, *collections* or *reports*. As all computed analytical data are stored permanently in a repository, in order to increase efficiency, the managers are able to cache loaded data and release them, if they are no longer required at runtime. Some managers are also able to postpone and aggregate update operations, but the consistency of computed data are still guaranteed.

Shared components. Shared components are instantiated only once in a running *Analyzer* application and are used by all opened projects together. Passing over auxiliary components, the most important one is a *launcher*, which is responsible for executing *tasks* over all such projects. *Tasks* represent

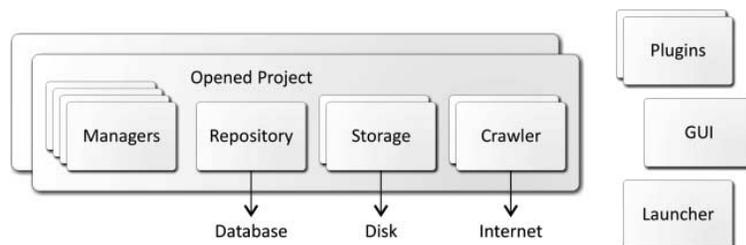


FIGURE 1. Architecture of *Analyzer*.

small units of analytical or other actions and computations. For example, each download of a selected file or computation over a given document is internally encapsulated and processed in a form of a task. Once it is decided that some work should be done, a new task is created, scheduled and later on prepared for execution, if no blocking dependencies exist. The execution itself is invoked by the *launcher*, maintaining a parallel environment with worker threads prepared in a pool. If a given task could not be successfully finished (for whatever reasons), launcher attempts to execute it repeatedly with a defined number of attempts.

Clearly, this model is a compromise, since it decreases the efficiency, but enables nearly full control over project processing. The user is able to attach/detach a particular project to/from the launcher and, thus, say whether tasks from the project should be executed at the moment, or not. As a consequence, the user can pause started computations and resume them later.

Graphical user interface. The GUI is based on possibilities of the NetBeans platform [38]. It brings the complete and robust environment for creating and managing projects and performing analyses from their configuration to browsing of the computed reports.

The browser itself contains adjustable windows through which the user is able to monitor the progress of computations and browse all existing entities in opened projects. The data are provided mainly in the form of interactive *trees* or *listings*. More complicated project actions, like creation of new components or configuration of analyses, are implemented using *wizards*.

3.2. Analysis model

In this section, we describe the life cycle of a typical analysis. As an example, suppose we want to develop an indexing service over Linked Data [39] and we want to know their basic characteristics, e.g. maximum and average depth or average number of child elements, to optimize our service. We choose DBpedia¹ as a source of data, create a project and set up all components described in the previous section. Documents are stored locally and the computed results in a MySQL database. For this analysis, we discard all documents except for RDF ones. Additionally, we add plugins to get expected results which have to be able to identify the RDF files, repair them and compute expected characteristics. When *Analyzer* downloads the files, the analyzed measures are computed and stored in the repository. Then we create a collection with all documents, *Analyzer* computes aggregated results and we can use the results for the optimization.

Later on, we decide to extend the service to index over another domain, e.g. data.gov.uk. We can use our old project and download and analyze a new version of the documents from

DBpedia and data.gov.uk. We add a new collection with newly added files from DBpedia to get the changes and a collection with all actually downloaded files to get actual characteristics of the examined domains.

From the previous example, we can derive the following list that represents a standard life cycle of each project:

- (i) Creation of a new project and configuration of repository, storages and crawlers,
- (ii) Selection and configuration of analyses using available plugins,
- (iii) Insertion of documents to be analyzed through import or download sessions,
- (iv) Computation of analytical results over documents of a given relative age,
- (v) Selection and configuration of clusters and collections,
- (vi) Document classification and assignment into collections and
- (vii) Computation of final statistical reports over particular collections.

Projects encapsulate inserted documents, configuration of analyses and computed data and represent a single research intent. During creation of a new project, the user can (besides changing other configuration) optionally select document types the project should be dedicated for. This selection is defined by a set of regular expression patterns over types.

The next step is a selection of *analyses* to be used in a given project (Fig. 2). This selection is based on a list of all currently available *plugins*, but the user can only instantiate a particular plugin once, if such a plugin is not configurable. As will be explained later, plugins offer their functionality through methods. An integral part of configuration of analyses is also the definition of the desired ordering of such methods.

Next, the user can import or download required documents. If a project involves type filtering, a given document is removed, if it does not match any of the provided patterns. The following step is creation of new clusters with sets of collections. Once a new collection is created, the classification of all existing documents satisfying other filtering criteria is automatically initiated. The final step is closing of clusters which invokes aggregation of results into reports. The reports are stored permanently and the user can browse them any time later.

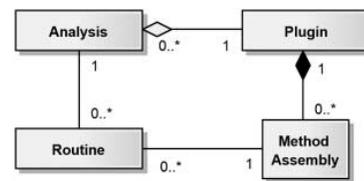


FIGURE 2. Analysis model diagram.

¹<http://dbpedia.org>.

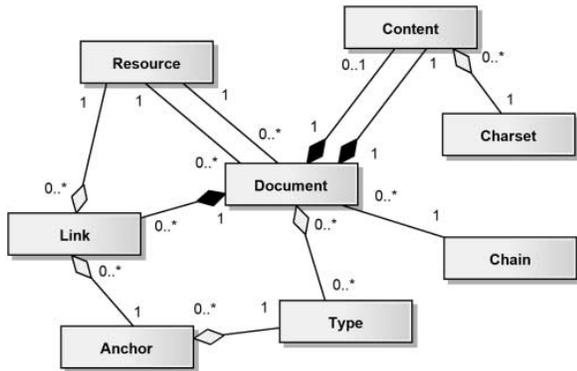


FIGURE 3. Document model diagram.

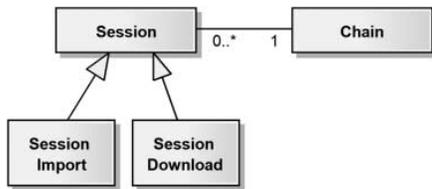


FIGURE 4. Session model diagram.

Model of documents. Each document to be analyzed is characterized by a pair of physical and logical *resources* (Fig. 3). The first one is the URL address from which the file was really imported, downloaded or sought from. The second one represents an address *Analyzer* ‘thinks’ the original file should be located at. However, the guessing heuristic is not currently completely implemented and is a subject of our future work which will exploit and combine the crawling module (Section 4), and the query analysis (Section 7). *Analyzer* itself is able to maintain multiple versions of the same file. Several consecutive import or download *sessions* are always grouped together into *chains*, defining a relative *age* for all documents in them (Fig. 4). It is not allowed to have more than one document of the same logical resource and the same relative age in a project.

The document entity itself is only an abstraction of a file—data *content* of a given file is treated independently and maintained using storages. Once a new document is inserted into a project, the corresponding file content is bound with this document entity. When generating content corrections, a new content version is always created and the previous one is thrown away, unless the previous one is the original one.

During the first steps of document processing, document *types* are recognized. Because this detection is realized by plugins, the typing concept is not limited and developers are allowed to work with their own typing namespace. However, we have proposed to harness standardized MIME types. The most

important fact about types is that each document is described by a set of recognized types, not only a single type. The idea behind types is simple: plugins recognize types, plugins analyze only documents of selected known types and, finally, projects can be restricted to processing of selected types only.

Analyzer also supports detection and processing of *links* between documents. A *link* is a typed reference from a source document to a target document, e.g. schema declarations in XML documents or image source file definitions in HTML [40] pages. The system is able to automatically delay processing of a given document until all required and accessible target documents are present in a project too.

Model of collections. After all documents with the same relative age are inserted into a project, the phase of computation of *results* is initiated. Each result is a small piece of information computed by a configured plugin over a particular document (Fig. 5). It is assumed that results are not the goal of analyses, they are created in order to be aggregated over multiple documents later on, in a form of *reports* over *collections*.

Collections are introduced in order to allow grouping of documents, i.e. creating named sets of documents (Fig. 6). The process of particular document classification is, once again, semantically defined by a configured plugin itself. Despite this fact, the user is also able to filter documents using general criteria like, e.g. resource addresses or relative age restrictions. Since some collections can be mutually related (e.g. they classify documents into multiple categories using a shared set of criteria), *Analyzer* requires grouping of collections into *clusters*.

When the classification of all documents is done, computed results can be aggregated as previously outlined (Fig. 5). This is done separately in each collection and, thus, the generated *reports* are always derived only from documents that are members of a given collection. However, not all documents may be provided with required results and, therefore, involved in this aggregation.

3.3. Model of plugins

Analyzer itself provides a general environment for performing analyses over documents and collections of documents, but

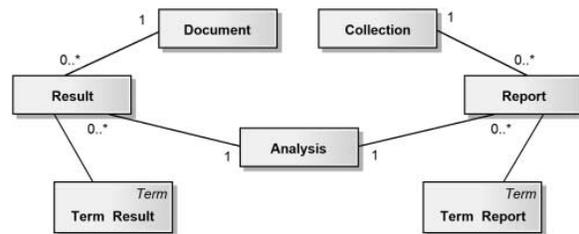


FIGURE 5. Result and report model diagram.

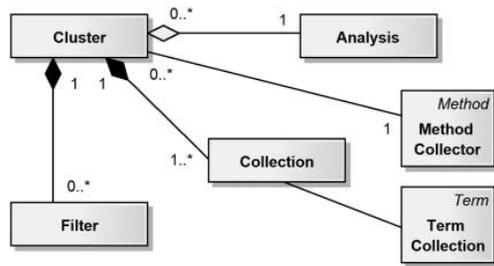


FIGURE 6. Collection model diagram.

the actual analytical logic is not a part of it. All analytical computations and mechanisms are implemented in plugins. The current distribution of *Analyzer* includes a few basic plugins for processing general files and XML-related files (see Sections 6 and 7), but the user is the one who is expected to create and use own extended plugins. It is also expected, but not required, that each plugin is determined for processing files of specific types. In other words, a plugin publishes its functionality and *Analyzer* makes it usable for analyses in projects.

Plugin methods. Disregarding the ability of a plugin to be configured (and thus, e.g. adjusted to particular analytical intents), each plugin specifies a set of document types that can be processed by it. This restriction is defined by regular expression patterns. The plugin functionality is provided through implemented *methods* (Fig. 7). They are of eight predefined types listed in the following enumeration. Although these methods are not Java methods but classes, we can omit this fact for simplicity.

- (i) *detector* recognizes types of a processed document,
- (ii) *racer* looks for outgoing links in a given document,
- (iii) *corrector* attempts to repair a content of a given document,
- (iv) *analyzer* produces results over a given document,

- (v) *collector* classifies documents into collections of a given cluster,
- (vi) *provider* creates reports by aggregating results of documents in a collection,
- (vii) *viewer* serves for browsing computed results over a document and
- (viii) *performer* serves for browsing computed reports over a collection.

After the user selects and configures all required analyses (plugins the user wants to use), the selection of particular available detectors, tracers, correctors and analyzers must be managed. This comprises not only the selection, but also the order of these methods. Despite different aims of these four method types, all of them may produce results. *Collectors* are methods that are responsible for classifying documents into collections. In other words, they make the decision, whether a given document belongs to a given related collection, or not. Once the user closes a cluster, *Analyzer* invokes *provider* methods over all its collections in order to aggregate results into reports. Finally, *viewer* and *performer* methods are used for presenting computed results over documents and computed reports over collections, respectively.

Execution of methods. The execution of tasks representing plugin methods is similar to the execution of other tasks, *Analyzer* only wraps the code written by the plugin programmer, invokes the computation and handles potentially raised errors or other forms of incorrect processing.

All plugin methods share the way how they access the functionality of *Analyzer* and how they acquire data about documents or other entities they are processing or generating. These requests are processed by *mediators*, objects with well-known interface and contract. Each method type works its with own specialized mediator, which allows only for relevant operations.

The mediator itself in fact only pretends to process of all these requests and internally simulates required actions, and the

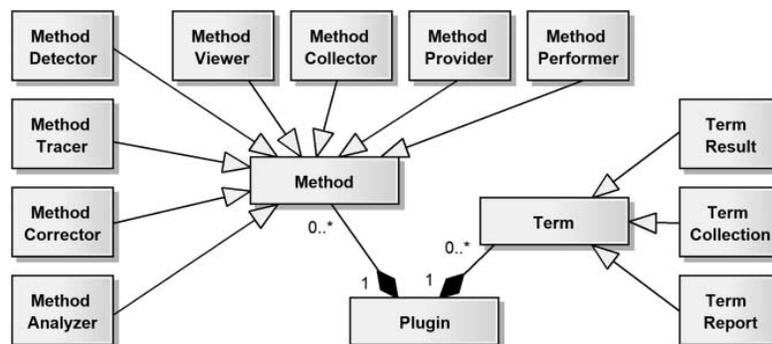


FIGURE 7. Plugin model diagram.

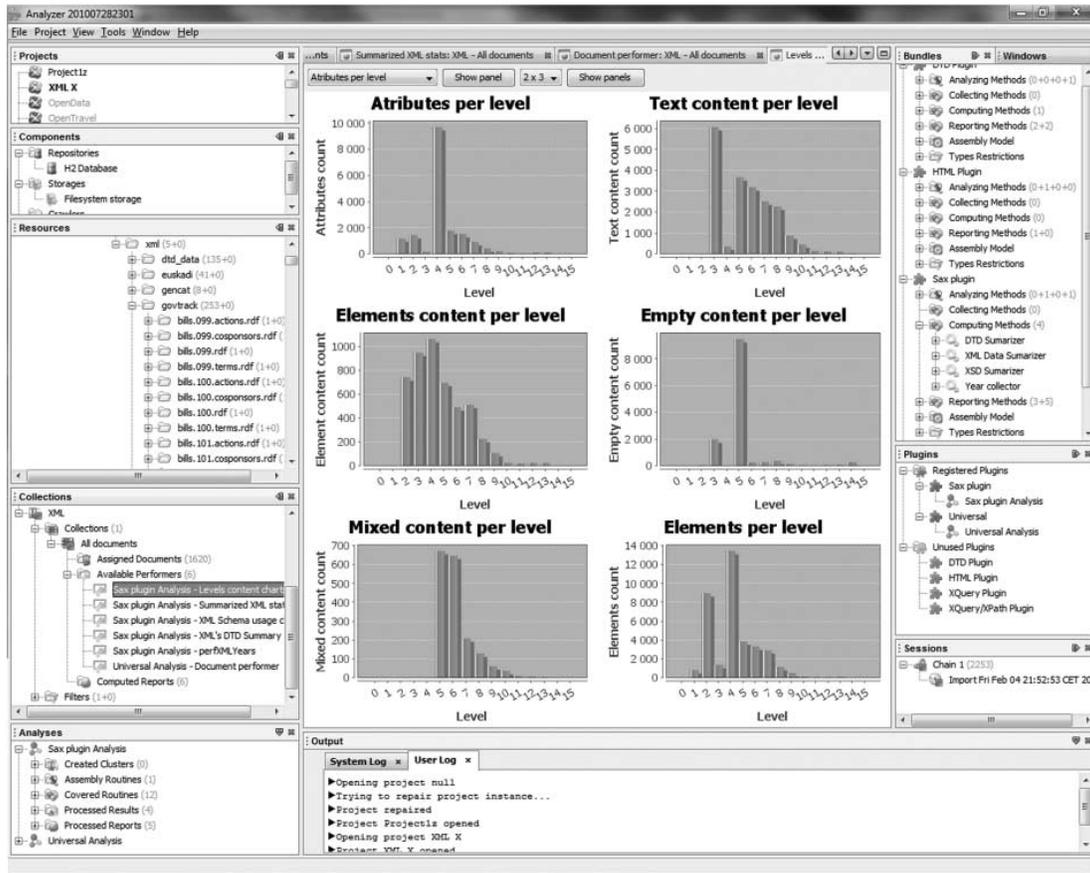


FIGURE 8. Sample screenshot of *Analyzer*.

real execution is postponed until the very end of a given task execution. As a consequence, we are able to reveal several forms of inconsistent behavior based on the violation of a published plugin contract.

Implemented plugins. The implementation of *Analyzer* comes with a few created plugins, which are ready for use. If we omit sample plugins demonstrating only framework possibilities, there are three main groups of plugins: a universal plugin for basic analyses of documents regardless of their types, a plugin for XML documents and their schema analyses (see Section 6) and, finally, a plugin for XQuery and XPath analyses (see Section 7).

3.4. Implementation of *analyzer*

Analyzer is implemented in Java 6 language [41] as a desktop application with a robust GUI. It is built on top of NetBeans 6.8 platform [38] and capable of the cross-platform usage.

The GUI of *Analyzer* is based on the possibilities of the NetBeans platform. It brings the complete environment for creating and managing projects and performing analyses from their configuration to browsing of computed reports. A sample screenshot image can be found in Fig. 8.

The default *Analyzer* distribution contains implementation of three repositories (MySQL server database through MySQL Connector 5.1.7 [42], embedded Apache Derby 10.5.1.1 database [43] and embedded H2 Database 1.1.117 [44]), two crawlers (simple built-in crawler and flooding Egothor 1.0 crawler [45]) and two storages (filesystem storage and dedicated Egothor storage).

3.5. Performance experiments

Apparently, the key performance role is represented by repositories and, therefore, the main impact brings primarily the number of analyzed documents. We configured two simple analyses with four methods for generating results in total.

TABLE 1. Performance characteristics.

Set name	Document count and size	Repository database	Document import	Results computation	Collections filling	Reports computation
A	1000 × 100 kB	Derby	7 s	60 s	14 s	12 s
		H2 DB	2 s	12 s	6 s	1 s
		MySQL	3 s	19 s	9 s	<1 s
B	10 000 × 10 kB	Derby	45 s	13 min	5 min	11 min
		H2 DB	10 s	100 s	90 s	60 s
		MySQL	15 s	135 s	70 s	10 s
C	100 000 × 1 kB	H2 DB	1 min	150 min	150 min	16 h
		MySQL	3 min	22 min	14 min	1 min

The documents were inserted into a project using import without copying of physical files. Finally, a single cluster was created and all processed documents were classified into its two of six collections.

Table 1 shows results of performed experiments over three different sets of documents. All tests were executed using a PC with Intel Core 2 Quad Q9550 2.83 GHz processor, 4 GB RAM and Gentoo Linux 10.1 operating system. The analyzed documents were stored on a local hard drive and also all three repositories stored their internal data locally on the same drive. MySQL Community Server 5.0.84 was installed locally and used through JDBC connector 3.0. Both filling of collections and computation of reports phases are based only on querying of the repository (document contents are never read).

It is worth noting that the purpose of these experiments was to show capabilities of the framework itself and not particular plugins, e.g. for XML documents analyses. Therefore, we used special plugins with methods of constant time complexity only. As we can see, there is a significant difference especially between H2 and MySQL. This can be explained by the inability of H2 to work with defined auxiliary index structures during selection queries.

As we have mentioned, there are four key aspects of the analytical process performed by *Analyzer*—data crawling, data correction, structural analysis and query analysis. In the following four sections we discuss them in detail. For each of the four topics we briefly describe the state of the art of the respective area and then we provide a description of the particular decisions and especially contributions applied in *Analyzer*.

4. DATA CRAWLING

The first key aspect of every data analysis is data gathering. As we have mentioned, *Analyzer* supports several types of input of the analyzed data, whereas the interesting aspect is the possibility of data crawling. The development of XML is closely tied to the Web; therefore, the Web is expected to contain vast amounts of XML-related data. Nevertheless, collecting the data

are surprisingly difficult. While there are a number of crawlers used to collect data from the Web, most of them are limited to HTML and widespread text-document formats like PDF [46].

XML-related data, which we consider in the broadest sense in this section, include all XML-based formats (including, e.g. XML Schema and XSLT) and related non-XML languages (like DTD or XQuery). When crawling the Web, some documents of these types may be found linked from HTML pages; however, others are referenced from the *primary documents*, like their schemas or included documents. Thus, the *secondary documents* cannot be located using HTML-based crawling—instead an XML-aware crawler must be able to parse both XML documents and the related formats (DTD etc.) to extract the links to the secondary documents.

Crawling the Web correctly is a difficult task, entangled within performance bottlenecks and surrounded by ethics and copyright rules. Creating a new crawler from scratch is apparently a senseless effort. Thus, we have opted to adapt and extend an existing HTML-oriented crawler. For our purposes we have evaluated the following systems:

- (i) *Xyleme/Larbin* [47, 48] was included in our list due to its ability to handle XML-to-DTD links. However, since Xyleme is not an open-source software, it might not be extended in our project.
- (ii) *Egothor* [45] was a system developed at the Charles University in Prague and, therefore, it was the first candidate for extension. Unfortunately, the community of Egothor developers is too small to ensure the required long life of the system.
- (iii) *Apache Nutch* [49] is an open-source project from the Apache family. Owing to its system of *extension points*, it originally seemed that the extension for XML-related data might be completely implemented using plugins.
- (iv) *Bixo* [50] is a *topical crawler* focused on mining data from selected locations, thus using a strategy different from traditional crawlers. However, it is tightly coupled with the mining methods.

- (v) *Google* and Google Web APIs [51] are included in our list as a kind of a benchmark because Google allows to search specific file types (like DTD) in its huge collection. Unfortunately, the exact method which Google uses for locating the specific documents is unknown.

As the previous list suggests, we selected the Apache Nutch system. We extended the system with a number of plugins and tuned its configuration toward the search for XML-related data. In addition, we had to modify the source code of Nutch at a few places. Our modifications and extensions together realized the following alterations to the original behavior of the crawler:

- (i) Improved address filtration—avoiding unwanted protocols (mailto, javascript) and formats (PDF, MP3 etc.).
- (ii) Altered document filtration—cutting out excessively large HTML documents, assuming that they would unlikely contain any XML-related links.
- (iii) *Whitelisting* apparently XML-related documents based on their reported MIME-type and (a part of) contents.
- (iv) *Blacklisting* unwanted documents. (Due to widespread errors in the Web content, we found blacklisting more efficient than whitelisting.)
- (v) Altered scoring mechanism—favorizing XML-related data in the download queue.
- (vi) Parsing XML-based documents and locating external references in them.

Our XML parser, based on a SAX implementation from the Xerces [52] family, is used to locate the following kinds of links in XML-based documents:

- (i) `schemaLocation` and `noNamespaceSchemaLocation` attributes in any XML document,
- (ii) `import`, `include`, and `redefine` elements in XSDs,
- (iii) `import` and `include` elements in XSLT programs,
- (iv) *processing instructions* in XML Style Sheets and
- (v) `include` and `externalRef` elements in RELAX NG [53].

Another important fact is that many web documents are malformed but still usable in crawling. Therefore, we made our parser robust with respect to non-well-formed data.

To depict the features of our modifications, Fig. 9a shows the percentage of document types encountered and downloaded during a testing run of unmodified Nutch system. The small non-XML part of these data are shown in detail in Fig. 9b—the left column corresponds to the behavior of the original Nutch system; the right column displays the results of the system after our modification. The figures are based on medium-scale tests—approximately 1 million documents of which 3.62% were XML-related.

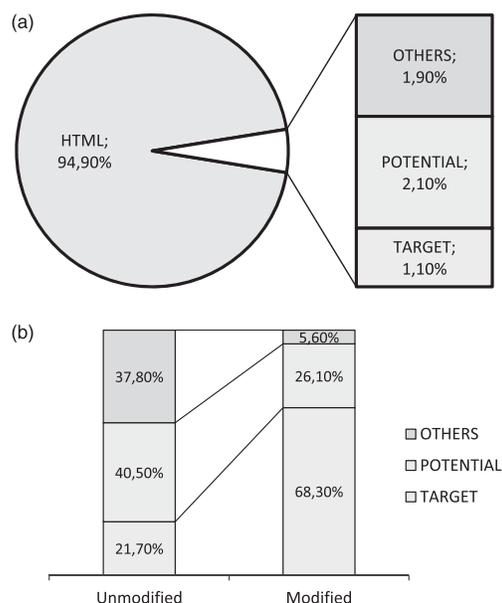


FIGURE 9. Results of experiments with data crawler. (a) Types of crawled documents. (b) Types of crawled documents after improvements.

5. PROCESSING OF INCORRECT DATA

Documents gathered by the automatic crawler as described in the previous section, or imported manually from different sources (e.g. filesystem), can contain several types of structural or semantic errors, which have to be solved. In *Analyzer*, we include error processing as the optional part of document processing. During this phase, *corrector* methods of available plugins are able to modify data contents of the documents, or discard the document from the analyzed set. Therefore, a plugin developer may propose methods for document correction and, thus, other methods can be assured that they are working only with correct data.

In the study of XML documents, we can witness a rather surprisingly high number of documents involving various forms of errors [11]. These errors can cause the documents to be not well formed, to not conform to the required structure or to have inconsistencies in data values. Anyway, the presence of errors causes at least obstructions and may completely prevent successful processing. Generally, we can modify existing algorithms to deal with errors, or we can attempt to modify invalid documents themselves.

We particularly focus on the problem of structural invalidity of XML documents. In other words, we assume the inspected documents are well formed and constitute trees; however, these trees do not conform to a schema in DTD or XML Schema, i.e. a *regular tree grammar* with the expressive power at the level of *single-type tree grammars* [22]. Having a potentially

invalid XML document, we process it from its root node toward leaves and propose minimum corrections of elements in order to achieve a valid document close to the original one. In each node of a tree we attempt to statically investigate all suitable sequences of its child nodes with respect to a content model and once we detect a local invalidity, we propose modifications based on operations capable of inserting new minimum subtrees, delete existing ones or recursively repairing them.

The remaining parts of this section present basic ideas of our correction model and proposed algorithms for finding structural repairs of invalid XML documents. Details of this proposal are presented in [15, 54].

5.1. Existing approaches

The proposed correction model is based primarily on ideas from [55, 56]. Authors of the former paper dynamically inspect the state space of a finite automaton for recognizing regular expressions in order to find valid sequences of child nodes with minimum distance. However, this traversal is not effective, requires a threshold pruning to cope with potentially infinite trees, repeatedly computes the same repairs and acts efficiently only in the context of *incremental validation*. Although these disadvantages are partially handled in the latter paper, its authors focused on document querying, but not repairing.

Next, we can mention an approximate validation and correction approach [57] based on testers and correctors from the theory of program verification. Repairs of data inconsistencies like functional dependencies, keys and multivalued dependencies are the subject of [58, 59].

Contrary to all existing approaches, we consider single type tree grammars instead of only local tree grammars. Thus, we work both with DTD and XML Schema. Approaches in [55, 57] are not able to find repairs of more damaged documents; we are able to always find all minimum repairs and even without any threshold pruning to handle potentially infinite XML trees. Next, we have proposed a much more efficient algorithm following only perspective ways of the correction and without any repeated repair computations. Finally, we have a prototype implementation [60] and performed experiments show a linear time complexity depending on a number of nodes in documents.

5.2. Proposed solution

Our correction framework is capable of generating local structural repairs for locally invalid elements. These repairs are motivated by the classic Levenshtein metric [61] for strings. For each node in a given XML tree and its sequence of child nodes we attempt to efficiently inspect new sequences that are allowed by the corresponding content model and that can be derived using the extended concept of measuring distances between strings. However, in our case we do not handle ordinary strings,

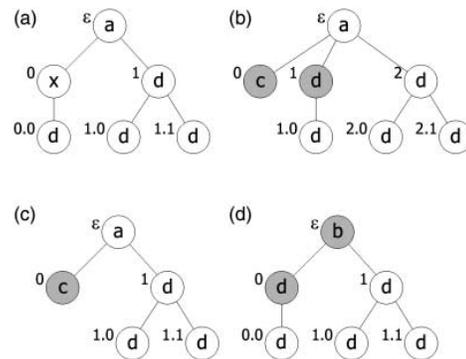


FIGURE 10. Sample XML tree with its three repairs. (a) Original tree. (b) Insert–Rename. (c) Rename–Delete. (d) Rename–Rename.

but sequences of nodes, which, in fact, are not only labels, but also entire subtrees.

The correction algorithm starts processing at the root node and recursively moves toward leaf nodes. We assume that we have the complete data tree loaded into the system memory and, therefore, we have a direct access to all its parts. Under all conditions the algorithm is able to find all minimum repairs, i.e. repairs with the minimum distance to the grammar and the original data tree according to the introduced cost function.

To illustrate the correction process throughout the following paragraphs, we use a sample XML document based on a fragment: `<a><x><d/></x><d><d/><d/><d/>`.

The derived data tree \mathcal{T} is depicted in Fig. 10a. Its underlying tree has nodes $\{\epsilon, 0, 0.0, 1, 1.0, 1.1\}$ and element labels are inscribed in nodes.

We want this document to conform to a simple local tree grammar \mathcal{G} , which requires that the label of the root node can only be a or b , child nodes of element a should match a regular expression $c.d^*$, elements b and d may contain an unlimited number of elements d , and, finally, element c should always be empty. Obviously, the sample data tree \mathcal{T} is not valid against \mathcal{G} , since element x is not allowed by the grammar at all.

Edit operations. Edit operations are elementary transformations that are used for altering invalid data trees into valid ones. They behave as deterministically defined functions, performing small local modifications with a provided data tree. Though the correction algorithm does not directly generate sequences of these edit operations, we can, in the end, acquire them using a translation of generated repairs, as will be explained later.

We have proposed and implemented the following edit operations:

- (i) insert a new leaf node,
- (ii) delete an existing leaf node,
- (iii) rename a label of a node,
- (iv) push a group of adjacent sibling nodes lower under a newly inserted internal node and

- (v) pull all sibling nodes one level higher deleting their original parent node.

Moreover, we have also formally studied other node and attribute operations. However, these operations were not yet fully implemented and, therefore, will be omitted in the rest of this section.

Update operations. Edit operations can be composed together into sequences. And if these sequences fulfill certain qualities, they can be classified as *update operations*. We have proposed

- (i) insertion of a new minimal subtree,
- (ii) deletion of an existing subtree and
- (iii) a recursive repair of a subtree with an option of changing a label of its root node.

Anyway, the purpose of each update operation is to correct a local part of a data tree in order to achieve its local validity. Unfortunately the correction algorithm does not generate these operations. The algorithm generates repairs based on repairing instructions, which are subsequently translated into sequences of edit operations. And this is the reason, why update operations are not defined deterministically similarly to edit operations. Having a particular sequence of edit operations, we can inspect its subsequences and if all required conditions are satisfied, a given subsequence can be viewed as an update operation of a corresponding type.

Assume that we have edit sequences

- (i) $\mathcal{X}_1 = (\text{addLeaf}(0, c), \text{renameLabel}(1, d))$;
- (ii) $\mathcal{X}_2 = (\text{renameLabel}(0, c), \text{removeLeaf}(0,0))$ and
- (iii) $\mathcal{X}_3 = (\text{renameLabel}(\epsilon, b), \text{renameLabel}(0, d))$.

Applying these sequences separately to data tree \mathcal{T} from our example, we obtain data trees depicted in Figs. 10b–d, respectively. If we use a unit cost function, these three data trees represent all minimal repairs of the original tree with the cost 2.

Repairing instructions. Assume that we are in a particular node in a data tree and our goal is to locally correct this node, which, passing over attributes, especially involves the correction of the sequence of its child nodes. Since the introduced model for measuring distances uses only non-negative values for the cost function, in order to acquire the global optimum, we can simply find minimum combinations of local optimums, meaning minimum repairs for all subtrees of original child nodes of the inspected one.

However, we need to find all minimum repairs, and since edit operations require particular positions in a current data tree to be specified, we cannot use them to describe all repairs. Assume, for example, that we have several options how to correct the first child node. If we delete it, all positions of nodes to the right must be shifted by one to the left, but if we accept the first child node, the original positions are preserved. Thus, we are not able

to use edit operations for describing multiple different repairing sequences.

The problem with the continuously changing numbers of positions is solved by the model of repairing instructions. We have exactly one instruction for each edit operation and these instructions represent the same transforming ideas, however, do not include particular positions to be applied on. Having a particular sequence of repairing instructions, we can easily translate it into the corresponding sequence of edit operations later on.

Correction intents. Being in a particular node and repairing its sequence of child nodes, the correction algorithm generally has many ways to achieve the local validity proposing repairs for all these involved nodes. As already outlined, these actions follow the model for measuring distances between ordinary strings. The Levenshtein metric is defined as the minimum number of required elementary operations to transform one string into another. These operations are insertion of one new symbol, deletion of an existing one and also replacement of an existing symbol with a new one. We follow the same model, however, we have edit and update operations and sequences of nodes. The given sequence can be viewed as an ordinary string over labels of its nodes. For example insertion of a new subtree at a given position stands for insertion of its label into the corresponding string of labels and, of course, recursive processing of such a new subtree.

The algorithm attempts to examine all suitable new words that are in the language of the provided regular expression restraining the content model of the inspected parent node. We do not generate word by word, but we attempt to inspect all these words statically using a notion of a correction and derived multigraphs.

Anyway, suppose that the algorithm has already processed the first few nodes from the inspected sequence of sibling nodes, and thus all nodes from the corresponding prefix of the original sequence are already involved in corrections. Now the algorithm must consider all possible actions that can be selected in order to involve at least one next node from the original sequence. The possibilities are modeled using the notion of correction intents.

In other words, the correction algorithm in each parent node has a variety of options how to achieve its validity, and particular steps performed with its child nodes are called *correction intents*, because we always examine one possible action from more permitted ones.

Correction multigraphs. All existing correction intents in a context of a given node can be modeled using a correction multigraph. Suppose that we need to process a sequence of child nodes with n nodes. This means that the graph will have $n + 1$ strata, numbered from 0 to n . Being on a stratum with number i , we have already processed right i first nodes from this sequence.

Each stratum is constructed from the Glushkov automaton for recognizing the provided regular expression restricting

the given sequence. This means that there are vertices corresponding to states of the automaton and directed edges reflecting the transition function in each stratum. Each such edge represents a new tree insertion operation. Similarly, we can define edges between strata to represent other allowed operations.

In other words, the correction multigraph represents all correction intents that can be derived for this sequence. And more precisely, each intent is represented by some edge in this multigraph. However, there can be intents that are represented by more edges at once.

To find best repairs for a provided sequence of nodes, we need to find all shortest paths in this multigraph, assuming that every edge is rated with an overall cost of corresponding nested correction intent associated with such an edge. However, to resolve these costs, we need to fully evaluate associated intents. And this represents non-trivial nested recursive computations. Anyway, we require that each edge can be evaluated in a finite time, otherwise we would obviously not be able to find required shortest paths.

If we return to our sample data tree \mathcal{T} , we can represent all nested correction intents derived for a root node a and a sequence $\langle x, d \rangle$ of its child nodes with respect to a regular expression $c.d^*$ by a correction multigraph in Fig. 11. For simplicity, edges are described only by abbreviated intent types (I for insert, D for delete, R for repair and N for rename), supplemented by a repairing instruction parameter if relevant and, finally, the complete *cost* of assigned intent repair. Names of vertices are concatenations of a stratum number and an automaton state.

Our goal is to find all shortest paths from the source vertex 00 to any of the target vertices 21 or 22 in the last stratum. Having found them, we can represent them in a form of the repairing multigraph in Fig. 12.

Repairs construction. Each correction intent can essentially be viewed as an assignment to the nested recursive processing. This model, in fact, has a transparent relation with a structure of an underlying tree itself and its processing from the root node toward leaves. The entire correction of a provided data tree is initiated as a special starting correction intent for the root node and processing of every intent always involves the construction

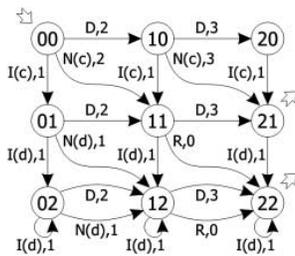


FIGURE 11. Sample correction multigraph.

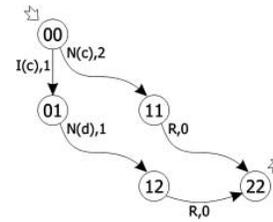


FIGURE 12. Sample repairing multigraph.

of at least the required part of the introduced multigraph with other nested intents. Therefore, we continuously invoke recursive computations of nested intents. When we reach the bottom of the recursion, we start backtracking, which involves gathering of found repairs. This means that after we have found the desired shortest paths at a given level, we encapsulate them in a form of a compact repair structure and pass it one level up, toward the starting correction intent.

Having found the shortest paths in the repairing multigraph for the starting intent, we have found repairs for the entire data tree. Each intent repair contains encoded shortest paths and related repairing instructions. Now we need to generate all particular sequences of repairing instructions and translate them into standard sequences of edit operations. Having one such edit sequence, we can apply it on the original data tree and we obtain its valid correction with a minimum distance.

Correction algorithms. Now we have completely outlined the model of the proposed correction framework. However, there are several related efficiency problems that would cause significantly slow behavior, if we would strictly follow this model. Therefore, we have introduced two particular correction algorithms. They both produce the same repairs, but there are key differences in their efficiency.

The first algorithm is able to directly search for the shortest paths inside each intent computation and, therefore, does not need the entire multigraphs to be constructed. The next improvement is based on caching already computed repairs using signatures distinguishing different correction intents, but intents with the same resulting repair structure. This causes this algorithm to never compute the same repair twice. The second algorithm is able to compute lazily even to the depth of the recursion. We have achieved this behavior by scattering all nested intents' invocation and multigraph edges' evaluation into small tasks, which are executed by a simple scheduler.

6. STRUCTURAL ANALYSIS

Having crawled and corrected the data, we can start with their analysis. In this chapter, we discuss several metrics, we implemented in the current version of *Analyzer*. We have to note that the implemented methods cover only basic characteristics

of XML, DTD and XSD, and they are based on previously published statistics of XML formats (see Section 8).

The implemented methods cover XML documents and the related schemas expressed in DTD and XML Schema. We created one plugin for each of the document types and one universal plugin to determine basic file attributes. The plugin for XML documents is applied to XML Schema documents, but beside the basic statistics, the usage of some specific constructs is measured.

We expected that the analyzed documents can be very large, and so we based the implementation of plugins on *Simple API for XML* (SAX) [62]. SAX allows for efficient work with large files, but, on the other hand, it complicates some analytic methods, e.g. XPath fragment search.

6.1. Common properties

Although we focused on three different formats, they have many common aspects which can be analyzed in a similar way. First, we analyze the number of entities used in the document, e.g. elements, attributes, declarations, etc. Secondly, we analyze the structural complexity of the used model. To define the structural characteristics of the document, we first need to define the XML document and XML schema. We use the definitions as presented in [63, 64]. The XML documents are expressed as ordered trees and XML schemas are expressed as regular expressions over element names.

DEFINITION 6.1. An *XML document* is a finite ordered tree $T = (\Sigma, N, E, r)$, where Σ is a finite alphabet, N is a set of nodes of the tree, E is a set of edges of the tree and $r \in N$ denotes a *root element* of the tree. Each node $\in N$ is associated with a *type of the node* which can be one of the following: *element, attribute, text, processing instruction or comment*. Nodes with element or attribute type are also associated with a node label $l \in \Sigma$ called an *element name* or an *attribute name*, respectively. The tree T is called Σ -tree

DEFINITION 6.2. A *DTD* is a collection of element declarations of the form $e \rightarrow \alpha$, where $e \in \Sigma$ is an element name and α is its content model, i.e. regular expression over Σ . The content model α is defined as $\alpha = \epsilon \mid p\text{cdata} \mid f \mid (\alpha_1, \alpha_2, \dots, \alpha_n) \mid (\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n) \mid \beta^* \mid \beta^+ \mid \beta^?$, where ϵ denotes the empty content model, *pdata* denotes the text content, f denotes a single element name, ‘,’ and ‘|’ stand for concatenation and union (of content models $\alpha_1, \alpha_2, \dots, \alpha_n$), and ‘*’, ‘+’ and ‘?’ stand for zero or more, one or more, and optional occurrence(s) (of content model β), respectively.

One of the element names $s \in \Sigma$ is called a *start symbol*.

In the following definitions we focus on the key characteristics of XML data. Due to space limitations and similarity, we provide definitions for XML schemas, in particular DTDs, and assume that their modifications for XSDs and XML documents are simple and apparent.

The complexity of the content model can be described by its depth.

DEFINITION 6.3. A *depth* of a content model α is inductively defined as follows:

- (i) $depth(\epsilon) = 0$;
- (ii) $depth(p\text{cdata}) = depth(f) = 1$;
- (iii) $depth(\alpha_1, \alpha_2, \dots, \alpha_n) = depth(\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n) = \max(depth(\alpha_i)) + 1; 1 \leq i \leq n$;
- (iv) $depth(\beta^*) = depth(\beta^+) = depth(\beta^?) = depth(\beta) + 1$.

Another important characteristic of the structure of whole schemas/documents are fan-out and fan-in.

DEFINITION 6.4. A *fan-out* of an element e is the cardinality of the set $\{f \mid e' \text{ is the element name of element } e, e' \rightarrow \alpha \text{ and the element name } f' \text{ of element } f \text{ occurs in } \alpha\}$.

DEFINITION 6.5. A *fan-in* of an element e is the cardinality of the set $\{f \mid f \rightarrow \alpha' \text{ and the element name } e' \text{ of element } e \text{ occurs in } \alpha'\}$.

Another metric of complexity of both documents and schemas is the usage of different types of content of an element. Generally, we can distinguish four types of content: empty, text, element and mixed. The former two are trivial; the latter two are defined as follows.

DEFINITION 6.6. A content model α is *element*, if $\exists e \in \Sigma$ such that e occurs in α .

DEFINITION 6.7. A content model α is *mixed*, if $\alpha = (\alpha_1 \mid \dots \mid \alpha_n \mid p\text{cdata})^* \mid (\alpha_1 \mid \dots \mid \alpha_n \mid p\text{cdata})^+$, where $n \geq 1$ and $\forall i$, such that $1 \leq i \leq n$, content model $\alpha_i \neq \epsilon \wedge \alpha_i \neq p\text{cdata}$. An element e is called *mixed-content* element if its content model α is mixed.

The last but not least important characteristic is the usage of specific structures. As mentioned in Section 2, a controversial characteristic are recursive elements.

DEFINITION 6.8. An element e is *recursive* if there exists at least one element d in the same document such that d is a descendant of e and d has the same element name as e .

6.2. XML documents

We created plugins to measure the following properties on XML documents:

- (i) The size of the XML document, e.g. in bytes, the number of elements or the number of attributes.
- (ii) Maximum depth of the document.
- (iii) Distribution of various types of content model over different levels.
- (iv) Recursion of elements.
- (v) Maximum and average fan-out.

- (vi) Usage of XML Schema versus DTD.
- (vii) Distinct element/attribute name usage.
- (viii) Namespace usage.

6.3. DTD analysis

For the DTDs, the system contains methods to compute the following statistics:

- (i) The size of the DTD, e.g. number of declarations of elements, attributes, notations, entities etc.
- (ii) Number of DTD-specific declarations of elements content type (`empty`, `any`, `'|'`, `'&'`, `'–'`).
- (iii) Number of DTD-specific declarations of attribute optionality (`#REQUIRED`, `#IMPLIED`, `#FIXED`).
- (iv) Usage of keys (i.e. attribute data types `ID` and `IDREFS(S)`).
- (v) Maximum, minimum and average depth.
- (vi) Average and maximum fan-outs and fan-ins.

6.4. XML schema analysis

The complexity of XSDs is basically measured in the same way as XML documents, since each XSD is at the same time an XML document. Besides these properties, we created a plugin for measurement of the usage of specific constructs as follows:

- (i) Type specification (`simpleType` and `complexType`).
- (ii) Restriction and extension of existing types.
- (iii) Content model of the elements (`sequence`, `choice`, `all`).
- (iv) Element groups and attribute groups (`group`, `attributeGroup`).

6.5. Results

Due to space limitations, we prepared a small sample of data to show the capabilities of the framework. A complete analysis, together with analysis of related operations (see Section 7) will be a subject of our very next future work and a separate paper. We used the current implementation of *Analyzer* with the basic implemented set of plugins. We run the application on a common dual-core processor with 2 GB RAM. The project was created with the H2 DB repository and the filesystem storage.

Data sources. For the experiments we used several data sources to get a variable sample of real-world data. As the open data became more popular, the methods for their processing are more required. Despite widespread use of XML, we can expect that the part of the data will be simple conversions from other formats like XLS² or CSV,³ and so we used also a small sample

of publicly available data to get some basic information about their structure. Firstly, the XML documents are gathered mostly from the U.S. federal executive branch datasets on `data.gov`. Some data were downloaded from the open data server of the Government of Catalonia⁴ and the rest of the documents from *U.S. congress*⁵ and *Open Data Euskadi*⁶. The datasets generally contain financial reports or geographical information related to the government. Secondly, we used the *OpenTravel* specification⁷ as a sample of XSD documents and compared their versions over the last 9 years.

XML documents statistics. In the first phase we took all data from all open data sources and computed global statistics over them. The results are shown in Fig. 13. The document type distribution by size is depicted in Fig. 13a and the type distribution by number in Fig. 13b. Having we gathered the XML documents, we can see that they are the main part of the data. On the other hand, the chart shows a big average size of RDF documents. Just 161 RDF documents take almost 40% of the total size. The distribution by size is illustrated in Fig. 13c and shows that the majority of this sample are documents between 10 kB and 10 MB.

To get a more precise picture of the examined sample, we computed the basic statistics over the documents. We focused on the number of the used elements and their attributes, depth and used schemas. The basic attributes of the examined files divided by the source are depicted in Table 2. The results show that the documents are generally flat; the average depth of `open.gov` sample exceeded depth of 10 with a maximum depth of 15. We can also see that XSDs are used only in the sample from `open.gov`. DTDs are partially used in documents from `euskadi.net`.

According to the results we can say that a typical open data document is shallow and its size is up to 5 MB. Anyway, due to the sample size and the limitations of space we cannot make a general conclusion. We shall focus on a detailed analysis over various large datasets in our future work.

XML schema statistics. In the second experiment we focused on XSDs used in the *OpenTravel* specification. The dataset contains 4637 documents with a total size of 194 MB. The dataset consist of several versions of the specification. In our experiment, we tried to show the evolution of the specification in terms of the size of the documents and the complexity of used constructs. The results of the analysis are shown in Fig. 14. In particular, we focused on global characteristics like the number of distinct element names (see Fig. 14a), the average depth (see Fig. 14b) or the average number of elements (see Fig. 14c). The second group of the examined values are XSD-specific

²Microsoft Excel format.
³Comma Separated Values.

⁴<http://opendata.gencat.cat/en/dades-obertes.html>.
⁵<http://www.govtrack.us/data/rdf/>.
⁶<http://opendata.euskadi.net/>.
⁷<http://opentravel.org/>.

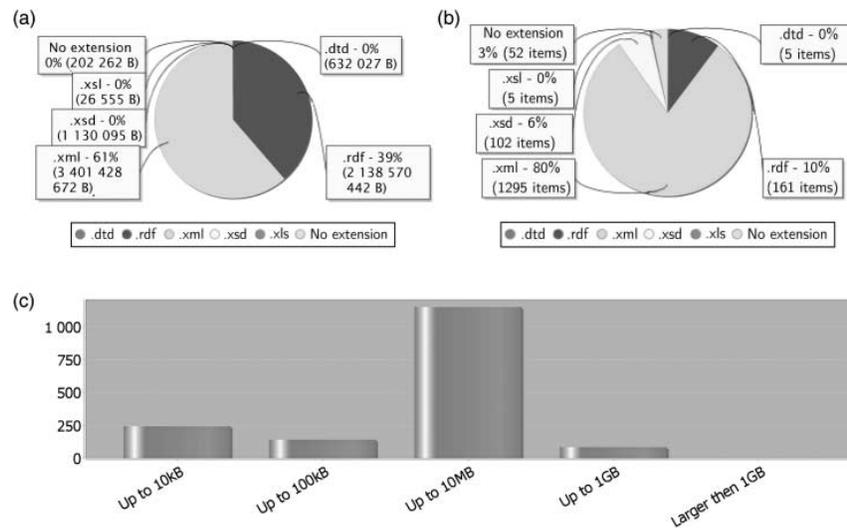


FIGURE 13. Document distribution. (a) Size distribution of types. (b) Type distribution. (c) Size distribution.

TABLE 2. Results of XML document analysis.

Source	Total size	Maximum size	Average size	Document count	Maximum depth	Average depth	Used DTD (%)	Used XSD (%)
open.gov	2.5 GB	67 MB	2314 kB	1156	15	10.16	0.2	88.6
gencat.cat	58.3 MB	40 MB	7.4 MB	8	11	6	0	0
govtrack.us	1.93 GB	77 MB	12.8 MB	253	5	4.76	0	0
euskadi.net	3.4 MB	1765 kB	27 kB	124	9	6.16	41.93	0
All	4643 MB	77 MB	2956 kB	1541	15	8.87	7.4	63.3

keywords. As an example, we show the evolution of the usage of sequences (see Fig. 14d), extensions (see Fig. 14e) and simple types (see Fig. 14f).

According to the results, we can see that, quite naturally, the specification is getting more complex and its depth is increasing. The largest change came between versions 2007A and 2007B. On the other hand, the usage of XSD constructs is stagnating; only the use of the `extension` keyword is rising, which reflects the general strategy of preserving backward compatibility.

7. QUERY ANALYSIS

Besides XML data and/or schemas, whose analysis was described in the previous section, analysis of queries over XML data may also give some insight into the way how XML is used. Such analysis will be useful for implementers of query engines or storage systems; in addition, the queries may also reveal useful facts on the data themselves (e.g. the presence

of recursion). There are a number of languages designed for processing and/or generating XML data. While the scientific community focuses on the XML Query (XQuery) language [5], real-world applications often use the XSLT [6]. These two languages, supported by their W3C standards, are currently the most widely used ones, although challenged by a number of more or less exotic languages like XDuca [65] etc.

XQuery 1.0 and XSLT 2.0 are powerful, Turing-complete [66] languages; however, their applications usually solve relatively simple problems like generating HTML pages or transforming XML between two schemas. Consequently, it is often believed that most applications use only small subsets of these languages. This observation is also supported by the fact that the most popular textbooks on XQuery or XSLT do not cover the languages exhaustively.

From the perspective of the implementor of an XQuery or XSLT processor, this observation suggests that a number of language features is rarely used and, therefore, not worthy of aggressive optimization. For example the *following/preceding* axes [4] are used significantly less frequently than the

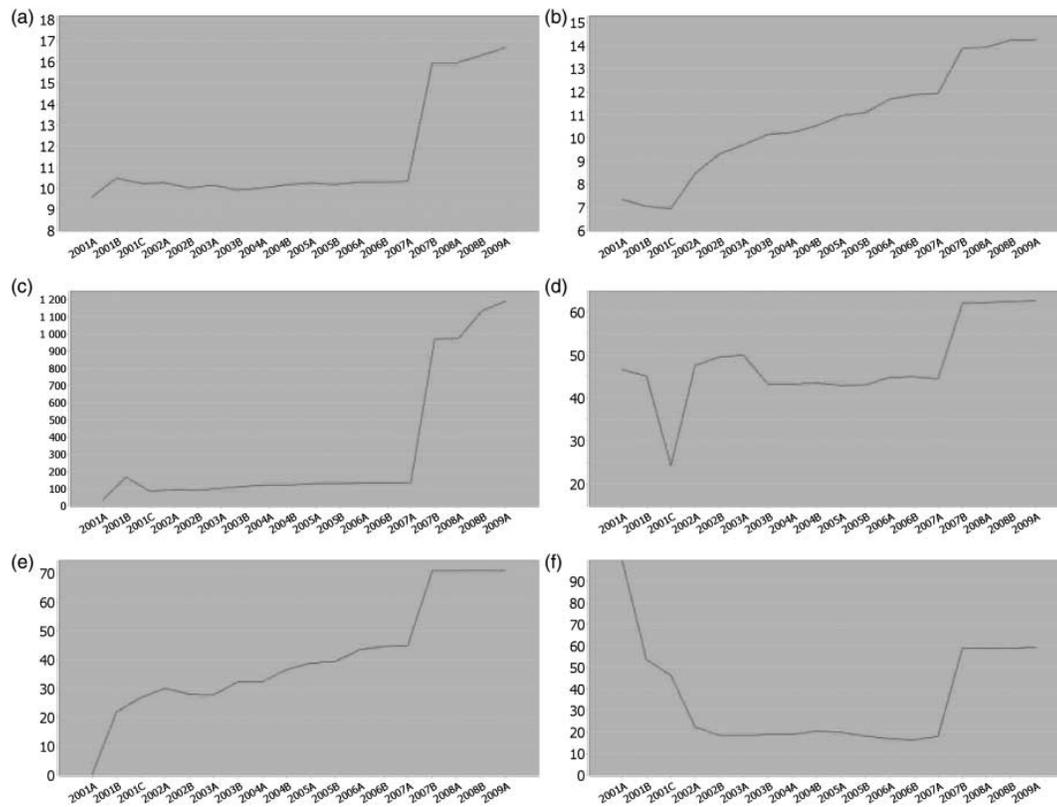


FIGURE 14. Results of XSD analysis. (a) Distinct element names. (b) Average depth. (c) Average number of elements. (d) Usage of sequences. (e) Usage of extensions. (f) Usage of simple types.

child/descendant axes; consequently, the majority of indexing and querying techniques like twig joins [67] are limited to the *child/descendant* axes.

Note that we introduced the observation with the clause ‘it is often believed’; indeed, it was probably never confirmed by any statistically significant study. Such a study was among our goals in this project. However, some of the tools required by this study were at least as interesting as the study itself. So, due to space limitations, similarly to the case of structural analysis we have decided to involve only an example of the query analysis, leaving a detailed version to future work and a separate paper.

In this section, we describe *XQAnalyzer*—a tool designed to support studies that include analysis of a collection of XQuery programs. Since it is a novel and unique part of the framework assumed to be widely used by the researchers, it can be used both as a standalone application and a plugin of *Analyzer*. *XQAnalyzer* consumes a set of XQuery programs, converts them into a kind of intermediate code and stores this *internal representation* in a repository (see Section 7.2). Subsequently, various analytical queries (see Section 7.1) may be placed on the repository to determine the presence or frequency of various

language constructs in the collection, including complex queries focused on particular combinations of constructs or classes of constructs.

The architecture of the *XQAnalyzer* is shown in Fig. 15. Each document from a given collection of XQuery programs is parsed and converted to the internal representation by the *XQConverter* component. The *XQEvaluator* component evaluates analytical queries and obtains statistical results.

7.1. Analytical queries

In the *XQAnalyzer*, the term *analytical query* denotes a pattern or a condition placed on an XQuery program, usually a search for a feature. Each XQuery program in the collection is evaluated independently, producing either a boolean value or a hit count representing the presence or number of occurrences in the program, respectively. The *XQEvaluator* then returns various statistical results like the percentage of programs that contain the searched feature or the histogram of hit counts over the repository.

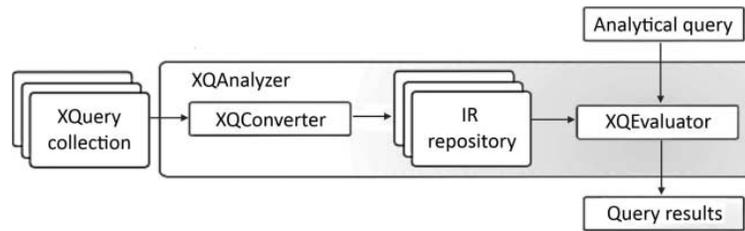


FIGURE 15. The architecture of the *XQAnalyzer*.

The language in which analytical queries are placed shall be powerful enough to allow sophisticated patterns like ‘call to a user-defined function placed inside a FLWOR statement whose arguments are independent of the FLWOR control variable’. At the same time, the potential users of the system must be able to learn the language quickly.

Given the fact that the tool is designed for research in the area of XML and, in particular, XQuery, the best choice would be a query language derived from XPath. XPath is naturally well known in the community and it is designed to place pattern-like queries on tree structures—in our case, a tree is a typical representation of a program during early stages of its analysis.

With the choice of XPath, the only remaining question in the design of the query language is the structure of the tree representing an XQuery program and its mapping to XML. In this representation, an analytical query is just an XPath query over the XML representation of query programs. The XML representation is discussed in the following section.

7.2. Internal representation of XQuery programs

The key issue in the design of *XQAnalyzer* is the internal representation of XQuery programs. In our approach, we do not want to limit the nature of the analytical queries; therefore, the internal representation must store any XQuery program without loss of any feature (perhaps except for comments). Furthermore, the internal representation is exposed to the user via the query interface; therefore, it should be as simple as possible. Finally, the internal representation affects the performance of the *XQEvaluator*. Since we already decided to use a tree-based representation queried through XPath, our freedom of choice is in the following issues:

- (i) The depth of the analysis performed before generating the internal representation.
- (ii) The ‘information density’ of the tree, i.e. the number and the degrees of nodes assigned to an individual language feature and, consequently, the size of the tree assigned to an input program.
- (iii) The representation of repeated components—either by recursion or by nodes with unlimited number of children.

- (iv) The names and attributes of nodes, including technical details regarding their mapping to the XML.

The W3C standards related to XQuery define at least the following two formalisms that might be used as a basis for our internal representation:

- (i) The *XQuery Grammar* defined in [5] using Extended Backus-Naur Form [68].
- (ii) The *Normalized XQuery Core Grammar* defined in [69] (also using EBNF).

Note that the XQuery formal semantics [69] is defined in terms of static/dynamic evaluation rules that may be considered as a kind of internal representation too. However, their application in our analytical environment would be impractically difficult.

There is also a number of formalisms defined in scientific literature. Among them, algebraic systems like XAT [70] might be easily adapted for our tree-based internal representation. However, these systems are always skewed toward a particular evaluation strategy (usually relational) and their use for other strategies would be difficult. For the same reason, we did not try to use *twig patterns* [67] in our representation.

Finally, it was proved [71] that each XQuery program may be translated to XSLT 2.0 [72]. Since each XSLT program is technically an XML document, the result of the translation may be used as an internal representation of the input XQuery program. Unfortunately, the conversion from XQuery to XSLT is not straightforward due to minor differences in the semantics of similar constructs (FLWOR vs. `<xsl:for>`). Furthermore, only a part of the XSLT syntax is expressed in terms of XML; the rest is hidden as XPath expressions inside the text of some XML attributes.

Among the existing formalisms mentioned so far, we have chosen the Normalized XQuery Core Grammar. The following reasons are behind this decision:

- (i) It is a part of the standard, and therefore is well known and not skewed toward any evaluation strategy.
- (ii) It is smaller than the full XQuery Grammar and it hides the redundant features of the XQuery language.

With respect to the depth of the analysis, the Normalized XQuery Core Grammar requires only parsing and normalization. In the canonical XQuery-processing chain, it would be followed by the optional static type analysis and the dynamic evaluation phase. Since static type analysis produces only additional information to augment the existing tree, it does not influence our selection of internal representation.

Of course, in real-world XQuery processors, normalization is followed by conversion to an algebra or other representation. As we have discussed above, these representations, if ever published, are hardly suitable for a strategy-independent tool.

The Normalized XQuery Core Grammar defines the *concrete* syntax of the XQuery Core. Therefore, it must define syntactic properties like priority and associativity of operators. Consequently, derivation trees constructed from this grammar have long branches containing semantically useless levels.

For analytical purposes, a more abstract representation is required, in a form of an *abstract syntax tree* (AST) [73]. An AST is present in almost every compiler; however, the corresponding *abstract* syntax grammar is rarely published or even standardized. In our case, we need an abstract grammar as close to the Normalized XQuery Core Grammar as possible. Therefore, we decided to start with the Normalized XQuery Core Grammar and to remove a part of the non-terminals corresponding to semantically useless levels that served only to define concrete syntax, collapsing the surrounding rules together. In a few cases we renamed the remaining non-terminals to more appropriate names (like *Operator*). The final set of non-terminals is listed in Table 3. When our internal representation is presented in the form of an XML document, these non-terminals become *XML elements*.

The rest of the semantic information is enclosed in *XML attributes* attached to the elements. These attributes contain either data extracted from the source text (like names of variables or contents of literals) or additional semantic information (like the axis used in an XPath axis step). In addition to these data required to preserve the semantics, we also added attributes that may help in recovering the original syntax before the normalization to XQuery Core (e.g. whether the abbreviated or the full syntax was used in axis step).

For example the XPath/XQuery expression

```
//car[@type="SUV"]
```

is normalized to the following XQuery Core expression

```
/descendant-or-self::* /child::car
  [attribute::type="SUV"]
```

and then converted to the internal representation shown (in the XML form) in Fig. 16.

The original form before normalization is described using the attribute `abbreviated="true"` in the *Axis* elements. The internal representation then may be queried using XPath expression like

TABLE 3. The elements of the internal representation.

Element	Use cases (%)	Test suite (%)
AtomicType	0.27	2.49
Axis	14.28	4.79
BaseURIDecl		0.04
BindingSequence	3.62	1.11
BoundarySpaceDecl		0.07
CData		0.01
CaseClauses	0.02	0.03
CharRef		0.02
CommaOperator	0.04	2.04
ConstructionDecl		0.04
Constructor	3.36	2.07
Content	4.03	2.11
ContextItem	0.22	0.11
CopyNamespacesDecl		0.02
DefaultCase	0.02	0.03
DefaultCollationDecl		0.01
DefaultNamespaceDecl		0.12
ElseExpression	0.07	0.08
EmptyOrderDecl		0.03
EmptySequence	0.02	0.63
EntityRef		0.01
Extension		0.03
FLWOR	1.97	0.79
ForClause	2.10	0.59
FunctionBody	0.40	0.16
FunctionCall	5.77	17.11
FunctionDecl	0.40	0.16
Hint	0.38	0.01
IfExpr	0.07	0.08
InClauses	0.27	0.15
KindTest	4.83	0.80
LetClause	1.25	0.32
Literal	4.61	20.32
ModuleDecl	0.07	0.00
ModuleImport	0.07	0.03
Name	4.21	2.14
NameTest	10.14	4.08
NamespaceDecl	0.20	0.18
OperandExpression	0.02	0.03
Operator	3.85	8.43
OptionDecl		0.01
OrderedExpr		0.01
OrderingModeDecl		0.02
Path	10.02	2.51
PragmaList		0.03
QuantifiedExpr	0.27	0.15
QueryBody	1.83	10.70
ReturnClause	2.04	0.90
SchemaImport	0.38	0.17
String	6.82	2.12
TestExpression	0.34	0.23

TABLE 3. Continued.

Element	Use cases (%)	Test suite (%)
ThenExpression	0.07	0.08
TupleStream	1.97	0.79
Type	0.98	2.62
Typeswitch	0.02	0.03
UnorderedExpr		0.01
ValidateExpr		0.02
VarDecl		2.42
VarRef	8.68	3.47
VarValue		2.42

TABLE 4. Axis usage.

Element	Use cases (%)	Test suite (%)
Child	71.63	82.67
Descendant		0.21
Attribute	5.33	3.70
Self		0.36
Descendant-or-self	23.04	10.40
Following		0.44
Parent		0.50
Ancestor		0.44
Preceding-sibling		0.42
Preceding		0.42
Ancestor-or-self		0.44

```

<Path initial-step="root">
  <Step>
    <Axis abbreviated="true" direction="forward" kind="descendant-or-self">
      <KindTest kind="any-kind"/>
    </Axis>
  </Step>
  <Step>
    <Axis abbreviated="true" direction="forward" kind="child">
      <NameTest name="car"/>
    </Axis>
    <Predicates>
      <Operator class="comparison" name="equals" subclass="general">
        <Path initial-step="context">
          <Step>
            <Axis abbreviated="true" direction="forward" kind="attribute">
              <NameTest name="type"/>
            </Axis>
          </Step>
        </Path>
        <Literal type="string" value="SUV"/>
      </Operator>
    </Predicates>
  </Step>
</Path>

```

FIGURE 16. Internal representation of an XPath expression.

```

//Step[Axis[@kind="child"] and
Predicates/Operator[@name="equals" and
Path and Literal]],

```

which finds any child-axis step combined with a predicate based on equality between a path expression and a literal.

7.3. Results

Since there is no standardized collection of real-world XQuery programs yet (except for small benchmarks like XMark [74]), we have chosen two artificial collections associated to the W3C XQuery language specification: the *XQuery Use Cases* [75] and the *XQuery Test Suite* [76]. The Use Cases collection consists of 85 ‘text-book’ XQuery programs prepared to demonstrate the most important features of the language; the Test Suite collection contains 14 869 small XQuery programs created to cover all features (the remaining 252 files in the original collection contain intentional parse errors). Although the Test Suite collection is more than 100 times larger in terms of the number of files, the real ratio of sizes (in terms of the number of AST nodes) is 31:1 because the Use Cases files are larger.

In Table 3 we show the frequency of core elements of the language, named according to the abstract grammar non-terminals derived from the Normalized XQuery Core Grammar. The percentages are defined by the number of occurrences divided by the total number of AST nodes in the collection (which was 4469 for the Use Cases and 138 949 for the Test Suite).

Besides the obvious difference between the two collections, corresponding to their purpose, there are the following noticeable observations: The frequency of quantified expressions (some or every) is about eight times smaller than the frequency of *for*-expressions. The *if*-expression is quite rare—once per 30 *for*-expressions or 50 operators. The number of features like *ordered/unordered*-expressions are omitted in the Use Cases. While frequent in the Test Suite, the comma operator is surprisingly rare in the Use Cases.

Table 4 shows the use of the twelve XPath axes. The percentages represent the frequency of individual axes among all axis step operators in the collection (which was 638 for the Use cases and 6623 for the Test suite). Note that the results correspond to the traditional belief that many axes are extremely rare.

8. RELATED WORK

As we have mentioned in the introduction, *Analyzer* is a quite a unique tool in the area of analyses of both XML data and general data types. Not to mention the area of query analyses, where there currently exists neither such a framework, not even results of a respective analysis.

Considering the area of XML data analysis, we can find several papers which involve the results of various types of statistical data analyses. However, in all the cases the respective tool (or a set of tools) is not available, so the analyses are neither extensible, nor repeatable. The papers analyze either the structure of DTDs, the structure of XSDs, the structure of XML documents (regardless their schema), or the structure

of XML documents in comparison with XML schemas. The sample data usually differ, whereas, since the authors did not use an advanced crawler, the set is usually quite small and unnatural.

DTD analyses. For the first time an analysis of the structure of DTDs, in particular 12 real-world DTDs, probably occurred in paper [25] and it was further extended in papers [63] (60 DTDs) and [77] (2 DTDs). They focused especially on the number of (root) elements and attributes, the depth of content models, the use of mixed content, IDs/IDREFs and *attribute optionality* (i.e. #IMPLIED, #REQUIRED and #FIXED), non-determinism and ambiguity. A side aim of the papers was a discussion of shortcomings of DTDs, since the XML Schema was only in the status of a preliminary working draft. The most important findings are that real-world content models are quite simple (the depth is always <10), the number of non-linear recursive elements is high (they occur in 58% of all DTDs), the number of shared elements is significant, and that IDs/IDREFs are not used frequently.

XML schema analyses. With the arrival of XML Schema, as the extension of DTD, a natural question has arisen: Which of the extra features of XML Schema not allowed in DTD are used in practice? Paper [21] is trying to answer it using statistical analysis of real-world XML schemas, in particular 109 DTDs and 93 XSDs. The most exploited features seem to be restriction of simple types (found in 73% of schemas), extension of complex types (37%) and namespaces (22%). The first finding reflects the lack of types in DTD, the second one confirms the naturalness of object-oriented approach, whereas the last one probably results from mutual modular use of XSDs. The other features are used minimally or are not used at all. The concluding finding is that 85% of XSDs define local-tree languages that can be defined by DTD as well. Paper [64], which also focuses directly on structural analysis of XSDs, defines 11 metrics of XSDs and two formulae that use the metrics to compute complexity and quality indices of XSDs. Unfortunately, there is only a single XSD example for which the statistics were computed.

XML data analyses. Paper [20] (and its extension [78]) analyzes the structure of about 200 000 XML documents directly, regardless of the eventually existing schema. The statistics are divided into two groups—statistics about the XML Web (e.g. clustering of the source websites by zones and geographical regions, the number and volume of documents per zone, the number of DTD/XSD references etc.) and statistics about the XML documents (e.g. the size and depth, the amount of markup and mixed-content elements, fan-out, recursion etc.). The most interesting findings of the research are that the structural information always dominates the size of documents, both mixed-content elements (found in 72% of documents) and recursion (found in 15% of documents) are important, and that documents are quite shallow (they have always fewer than eight levels in average).

A much simpler document analysis performed in paper [79] consists of two parts—a discussion of different techniques

for XML processing and an analysis of real-world XML documents. The sample data consists of 601 XHTML [80] web pages, three documents in the *DocBook* format,⁸ an XML version of Shakespeare's plays⁹ (i.e. 37 XML documents with a common simple DTD) and documents from the *XML Data Repository* project.¹⁰ The analyzed properties are the maximum depth, the average depth, the number of simple paths and the number of unique simple paths; the results are similar to the previous cases.

XML data vs. XML schema analyses. The work initiated in the previously mentioned articles is taken up by probably the latest paper in this field [11]. It enhances the preceding analyses and defines several new constructs for describing the structure of XML data (e.g. so-called *DNA* or *relational patterns*) and analyzes XML documents together with their eventual DTDs/XSDs that were collected semi-automatically, i.e. with interference of the human operator. The reason is that automatic crawling of XML documents generates a set of documents that are unnatural and often contain only trivial data which cause misleading results. The collected data consist of about 16 500 XML documents of more than 20GB in size divided into 133 collections, whereas only 7.4% have neither a DTD nor an XSD. Such a low ratio is probably caused by the semi-automatic gathering.

The data were divided into five categories—*data-centric*, *document-centric*, *exchange*, *report* and *research*. The first two categories correspond to classical categories [81], the other three are introduced to enable a finer division. The statistics described in the paper are also divided into several categories—*global* (e.g. number of various constructs), *level* (i.e. distribution of various constructs per level), *fan-out* (i.e. branching), *recursive* (i.e. types and complexity of recursion), *mixed-content* (i.e. types and complexity of mixed contents), *DNA* (i.e. types and complexity of DNA patterns) and *relational* (i.e. types and complexity of relational patterns). They were computed for each document category and, if possible, also for both XML documents and XML schemas, and the results were compared.

Most interesting findings and conclusions for all categories of statistics are partly expected (e.g. that tagging usually dominates the size of document or that mixed-content elements are used in 77% of document-centric documents) and partly similar to the previous results (e.g. that the average depth of XML documents is about 5). However, there are also some very interesting observations and conclusions. For example recursion statistics show that despite the typical assumptions recursion occurs quite often, especially, in document-centric (43%) and exchange (64%) documents; the number of distinct recursive elements is typically low (for each category <5); and that the type of recursion commonly used is very simple.

⁸<http://www.docbook.org/>.

⁹<http://www.ibiblio.org/xml/examples/shakespeare/>.

¹⁰<http://www.cs.washington.edu/research/xmldatasets/>.

As we can see, the performed analyses reflect the development of XML technologies and use of XML data in various applications. The problem is that all the papers are relatively old (the first paper is from 2001, the last one is from 2006), and so the results are obsolete. At the same time, there occur new XML technologies, data types and applications, whose analysis would be very useful. And a similar study would be even more useful in the area of XML operations.

9. OPEN PROBLEMS

Even though the current version of *Analyzer* is a fully functional system that can be applied on analyses of real-world XML data (as partially shown in Sections 6.5 and 7.3), there are naturally various open problems to be focussed on.

Advanced crawling. Apart from classical crawling strategies on the basis of URLs used in HTML or XHTML linking constructs, and detection of file types using file extensions or MIME types etc., we can exploit properties of the particular type of data more deeply. For instance XML documents involve references to respective XML schemas that they are supposed to be valid against; XQuery queries refer to the documents they are posed over, whereas XSDs can refer to other schemas they consist of using constructs such as `import`, `include` or `redefine`. Also more advanced linking XML technologies, such as XPointer [82] or XLink [83] can be used to mutually refer the data.

Even a more advanced crawler can deal with typical situations when the referenced data are not present directly in the given address, but ‘close’ to it, i.e. in a neighboring directory, in a file with a slightly modified name etc. In this case the search strategy cannot be exact, but some kind of fuzzy searching and ‘guessing’ must be incorporated. A similar situation occurs in case of, e.g., XQuery queries, where we usually know the exact name of the queried document, but not the path.

The last situation to be solved occurs in situations when a given file type (e.g. a script with XPath queries) does not have a specific extension or a user does not know and use it. So the crawler cannot rely on the extensions and/or other simple types of identification and must analyze directly the content of the file. Naturally, such analysis cannot be detailed since it would highly worsen the efficiency of crawling, but a kind of reasonable heuristics for particular file types must be proposed and, in particular, tested.

Improvements of analytical plugins. The current analytical plugins are able to analyze basic structural aspects of XML documents and XML schemas. Naturally, they can be further extended so that they cover most of the statistics used in the related work (see Section 8). However, since the current approaches are relatively old, we can go even further and analyze new, not considered or advanced features. An example can be new constructs of *XML Schema 1.1* [84, 85] such as, e.g. `assert` and `report` that enable to express advanced

integrity constraints using XPath, XSLT scripts and their constructs, complexity and expressive power, or advanced schema languages, such as *Schematron* [86] or *RELAX NG* [53].

Detailed analysis of current real-world data. Having such a robust tool for analysis of real-world data, a natural next step is to perform a detailed analysis of the current real-world data. Considering the XML data we have focussed on in our first use case implemented in the plugins, we can proceed in several steps. First, a detailed analysis that would cover all the metrics and observations from the existing papers on data analyses (as described in Section 8) can be performed, whereas the found differences would bear highly useful and interesting information.

In the second step, and in combination with the previous open issue of advanced crawling, we can focus on analysis of real-world XML operations, in particular XPath and XQuery queries. To our knowledge, there exist no such results, while, on the other hand, the knowledge of typical queries used in the real-world applications would highly help in the respective optimization strategies. An interesting target analysis would also be use of the queries within other XML technologies, e.g. XPath queries in XSLT scripts or XSDs, XSDs in Web Services [7] etc.

And, last but not least, an important aspect of statistical analyses of real-world data, not just that of XML one, is analysis of their evolution. A periodical, e.g. monthly, report of results and their aggregation would bear even more important information on evolution and tendencies of XML applications and, hence, could be used for more advanced optimization purposes.

Analysis of other kinds of data. Despite the fact that XML data still keep a leading role in data representation and the related XML technologies are robust and mature, there exist other important formats and data types that are becoming more popular. A classical example are data types related to the Semantic Web [87], such as RDF triples [9], ontologies [88], Linked Data [39] etc. In this case we need to solve similar issues, i.e. crawling, correction and analyses, whereas other aspects, namely evolution, are even more important.

10. CONCLUSION

The main aim of this paper was to introduce a complex, open and extensible system called *Analyzer* and to describe several related research problems, which we have focussed on. *Analyzer* allows for performing the full process of data analysis that consists of the following steps:

- (i) data crawling,
- (ii) data correction,
- (iii) application of analyses and
- (iv) aggregation and visualization of results.

As a first use case we implemented and tested modules for analyses of real-world XML documents, XML schemas and

XQuery queries. In the first three steps of the process we focussed in more detail especially on issues of efficient crawling of XML data, re-validation of invalid XML documents, exploitation of the similarity of XML data in data analysis and XML query analysis.

Despite our original motivations related to XML technologies, we finally implemented an application that is completely capable of performing analyses over documents of whatever types. *Analyzer* represents a framework that gives a user an environment for gathering documents, configuring analyses, managing and scheduling computations, permanent storage for files and computed data, and a browser for presenting generated reports. The key advantages of *Analyzer* are as follows:

- (i) multiple versions of the same document are supported,
- (ii) documents can be described by multiple types concurrently,
- (iii) automatic attempts to download referenced documents are performed,
- (iv) projects can be forced to process only documents of selected types,
- (v) all analytical logic is implemented separately in plugins,
- (vi) executing scheduled tasks in multi-threaded environment is exploited,
- (vii) started computations can be interrupted and resumed later and
- (viii) computed data are permanently stored and available for browsing.

Our future plans will primarily be targeted to issues discussed in Section 9. First, we will focus on further improvements of existing plugins related to XML data analyses and their exploitation in throughout analysis of both current state of real-world XML documents and evolution of XML data in the following months. We plan to repeat the analysis monthly and publish the new as well as aggregated results on the Web. We believe that such a unique analysis will provide the research community with important results useful for both optimization purposes as well as development of brand new approaches. Concurrently, we shall shift our target area to the new types of data such as RDF triples, Linked Data, ontologies etc.

ACKNOWLEDGEMENTS

This work was partially supported by the Czech Science Foundation (GAČR), grants number P202/10/0573 (J. Stárka), 201/09/P364 (I. Mlýnková), 201/09/0990 (D. Bednárek) and the Grant Agency of the Charles University (GAUK), grant number 410511 (M. Svoboda).

REFERENCES

- [1] Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E. and Yergeau, F. (2008) *Extensible Markup Language (XML) 1.0* (5th edn). W3C. <http://www.w3.org/TR/xml/>.
- [2] Thompson, H.S., Beech, D., Maloney, M. and Mendelsohn, N. (2004) *XML Schema Part 1: Structures* (2nd edn). W3C. <http://www.w3.org/TR/xmlschema-1/>.
- [3] Biron, P.V. and Malhotra, A. (2004) *XML Schema Part 2: Datatypes* (2nd edn). W3C. <http://www.w3.org/TR/xmlschema-2/>.
- [4] Clark, J. and DeRose, S. (1999) *XML Path Language (XPath) Version 1.0*. W3C. <http://www.w3.org/TR/xpath>.
- [5] Boag, S., Chamberlin, D., Fernandez, M.F., Florescu, D., Robie, J. and Simeon, J. (2010) *XQuery 1.0: An XML Query Language* (2nd edn). W3C. <http://www.w3.org/TR/xquery/>.
- [6] Clark, J. (1999) *XSL Transformations (XSLT) Version 1.0*. W3C. <http://www.w3.org/TR/xslt>.
- [7] Booth, D. and Liu, C.K. (2007) *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*. W3C. <http://www.w3.org/TR/wsdl20-primer/>.
- [8] Dahlstrom, E., Dengler, P., Grasso, A., Lilley, C., McCormack, C., Schepers, D., Watt, J., Ferraiolo, J., Jun, F. and Jackson, D. (2011) *Scalable Vector Graphics (SVG) 1.1* (2nd edn). W3C. <http://www.w3.org/TR/SVG/>.
- [9] Beckett, D. (2004) *RDF/XML Syntax Specification* (Revised edn). W3C. <http://www.w3.org/TR/rdf-syntax-grammar/>.
- [10] Oracle (2010) *OpenOffice.org Project*. Oracle. <http://www.openoffice.org/>.
- [11] Mlýnková, I., Toman, K. and Pokorný, J. (2006) Statistical Analysis of Real XML Data Collections. *Proc. 13th Int. Conf. Management of Data*, New Delhi, India, December 14–16, pp. 20–31. Tata McGraw-Hill Publishing, New Delhi, India.
- [12] Stárka, J., Svoboda, M., Sochna, J. and Schejbal, J. (2010) *Analyzer 1.0*. Charles University in Prague, Czech Republic. <http://analyzer.kenai.com/>.
- [13] Svoboda, M., Stárka, J., Sochna, J., Schejbal, J. and Mlýnkova, I. (2010) Analyzer: A Framework for File Analysis. *Proc. 2nd Int. Workshop on Benchmarking of Database Management Systems and Data-Oriented Web Technologies of the 15th Int. Conf. Database Systems for Advanced Applications*, Tsukuba, Japan, April 1–4, Lecture Notes in Computer Science 6193, pp. 227–238. Springer, Berlin/Heidelberg.
- [14] Sochna, J. (2010) Collecting XML data and meta-data from the internet (in Czech). Master Thesis, Charles University in Prague, Czech Republic. <http://www.ksi.mff.cuni.cz/~bednarek/dp/Sochna.pdf>.
- [15] Svoboda, M. (2010) Processing of incorrect XML data. Master Thesis, Charles University in Prague, Czech Republic. <http://www.ksi.mff.cuni.cz/~mlynkova/dp/Svoboda.pdf>.
- [16] Stárka, J. (2010) Similarity of XML data. Master Thesis, Charles University in Prague, Czech Republic. <http://www.ksi.mff.cuni.cz/~mlynkova/dp/Starka.pdf>.
- [17] Schejbal, J. (2010) A system for analysis of collections of XML queries. Master Thesis, Charles University in Prague, Czech Republic. <http://www.ksi.mff.cuni.cz/~bednarek/dp/Schejbal.pdf>.
- [18] Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D.J. and Naughton, J.F. (1999) Relational Databases for Querying XML Documents: Limitations and Opportunities. *Proc. 25th Int. Conf. Very Large Data Bases*, Edinburgh, Scotland, September 7–10, pp. 302–314. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

- [19] Runapongsa, K. and Patel, J.M. (2002) Storing and Querying XML Data in Object-Relational DBMSs. *Proc. Workshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, Prague, Czech Republic, March 25–27, pp. 266–285. Springer, London, UK.
- [20] Mignet, L., Barbosa, D. and Veltri, P. (2003) The XML Web: A First Study. *Proc. 12th Int. Conf. World Wide Web*, Budapest, Hungary, May 20–24, pp. 500–510. ACM, New York, NY, USA.
- [21] Bex, G.J., Neven, F. and den Bussche, J.V. (2004) DTDs Versus XML Schema: A Practical Study. *Proc. 7th Int. Workshop on the Web and Databases: Collocated with ACM SIGMOD/PODS 2004*, Paris, France, June 16–17, pp. 79–84. ACM, New York, NY, USA.
- [22] Murata, M., Lee, D. and Mani, M. (2005) Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Technol.*, **5**, 660–704.
- [23] Bex, G.J., Neven, F. and Vansummeren, S. (2007) Inferring XML Schema Definitions from XML Data. *Proc. 33rd Int. Conf. Very Large Data Bases*, Vienna, Austria, September 23–27, pp. 998–1009. ACM, New York, NY, USA.
- [24] Vošta, O., Mlýnková, I. and Pokorný, J. (2008) Even an Ant can Create an XSD. *Proc. 13th Int. Conf. Database Systems for Advanced Applications*, New Delhi, India, March 19–21, pp. 35–50. Springer, Berlin/Heidelberg.
- [25] Sahuguet, A. (2001) Everything you Ever Wanted to Know about DTDs, but were Afraid to Ask (Extended Abstract). *Selected Papers from the 3rd Int. Workshop WebDB 2000 on The World Wide Web and Databases*, Dallas, TX, USA, May 18–19, pp. 171–183. Springer, Berlin/Heidelberg.
- [26] Krátký, M., Pokorný, J. and Snášel, V. (2002) Indexing XML Data with Ub-Trees. *Proc. Advances in Databases and Information Systems*, Bratislava, Slovakia, September 8–11, pp. 155–164. Springer, Berlin/Heidelberg.
- [27] Krátký, M., Pokorný, J. and Snášel, V. (2004) Implementation of XPath Axes in the Multi-dimensional Approach to Indexing XML Data. *Proc. EDBT 2004 Workshops on Current Trends in Database Technology*, Heraklion, Crete, Greece, March 14–18, pp. 46–60. Springer, Berlin/Heidelberg.
- [28] Bayer, R. (1997) The Universal B-Tree for Multidimensional Indexing: General Concepts. *Proc. Int. Conf. Worldwide Computing and Its Applications*, Tsukuba, Japan, March 10–11, pp. 198–209. Springer, Berlin/Heidelberg.
- [29] Fenk, R. (2002) The Bub-tree. *Proc. 28th Int. Conf. Very Large Data Bases*, Hong Kong, China. Morgan Kaufman Publishers.
- [30] Gold, E.M. (1967) Language Identification in the Limit. *Inf. Control*, **10**, 447–474.
- [31] Bex, G.J., Gelade, W., Neven, F. and Vansummeren, S. (2008) Learning Deterministic Regular Expressions for the Inference of Schemas from XML Data. *Proc. 17th Int. Conf. World Wide Web*, Beijing, China, April 21–25, pp. 825–834. ACM, New York, NY, USA.
- [32] Bex, G.J., Neven, F., Schwentick, T. and Vansummeren, S. (2010) Inference of concise regular expressions and DTDs. *ACM Trans. Database Syst.*, **35**, 1–47.
- [33] Rahm, E. and Bernstein, P.A. (2001) A survey of approaches to automatic schema matching. *VLDB J.*, **10**, 334–350.
- [34] Wojnar, A., Mlýnková, I. and Dokulil, J. (2010) Structural and semantic aspects of similarity of document type definitions and XML schemas. *Inform. Sci.*, **180**, 1817–1836.
- [35] Yi, S., Huang, B. and Chan, W.T. (2005) XML application schema matching using similarity measure and relaxation labeling. *Inform. Sci.*, **169**, 27–46.
- [36] Bohannon, P., Freire, J., Roy, P. and Simeon, J. (2002) From XML Schema to Relations: A Cost-Based Approach to XML Storage. *Proc. 18th Int. Conf. Data Engineering*, San Jose, CA, USA, February 26–March 1, p. 64. IEEE Computer Society, Washington, DC, USA.
- [37] Klettke, M. and Meyer, H. (2001) XML and Object-Relational Database Systems—Enhancing Structural Mappings Based on Statistics. *Selected Papers from the 3rd Int. Workshop WebDB 2000 on The World Wide Web and Databases*, London, UK, pp. 151–170. Springer, Berlin/Heidelberg.
- [38] NetBeans 6.8 Platform. <http://platform.netbeans.org/>.
- [39] Bizer, C., Heath, T. and Berners-Lee, T. (2009) Linked data—the story so far. *Semant. Web Inform. Syst.*, **5**, 1–22.
- [40] Raggett, D., Hors, A.L. and Jacobs, I. (December 1999) *HTML 4.01 Specification*. W3C. <http://www.w3.org/TR/html401/>. (7 May 2011, date online accessed).
- [41] Sun Microsystems *Java 6 Standard Edition*. <http://java.sun.com/javase/6/>. (7 May 2011, date online accessed).
- [42] MySQL Connector 5.1.7. <http://dev.mysql.com/downloads/connector/j/>. (7 May 2011, date online accessed).
- [43] Apache Derby 10.5.1.1 Database. <http://db.apache.org/derby/>. (7 May 2011, date online accessed).
- [44] H2 Database 1.1.117. <http://www.h2database.com/>. (7 May 2011, date online accessed).
- [45] Galamboš, L. (2006). *Egothor 1.0, Java Search Engine*. <http://www.egothor.org/>. (7 May 2011, date online accessed).
- [46] (2008) *ISO 32000-1:2008: Document management-portable document format*. Adobe. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=51502. (7 May 2011, date online accessed).
- [47] Xyle, L. (2001) *Xyleme: A Dynamic Warehouse for XML Data of the Web*, Grenoble, France, July 16–18, pp. 3–7. IEEE Computer Society, Washington, DC, USA.
- [48] Ailleret, S. (2009). *Larbin: Multi-purpose Web crawler*. <http://larbin.sourceforge.net/>. (7 May 2011, date online accessed).
- [49] Cafarella, M. and Cutting, D. (2004) Building Nutch: open source search. *Queue*, **2**, 54–61.
- [50] Judd, D. and Groschupf, S. (2009). *Bixo—A Webcrawler Toolkit*. <http://bixo.101tec.com/wp-content/uploads/2009/05/bixo-intro.pdf>. (7 May 2011, date online accessed).
- [51] Mayr, P. and Tosques, F. (2005) Google Web APIs—an Instrument for Webometric Analyses? *Proc. 10th Int. Conf. Int. Society for Scientometrics and Informetrics*, Stockholm, Sweden, July 24–28.
- [52] Apache Software Foundation (2010). *Xerces Java parser*. <http://xerces.apache.org/xerces2-j/>. (7 May 2011, date online accessed).

- [53] Murata, M. (2002) *RELAX (Regular Language Description for XML)*. <http://www.xml.gr.jp/relax/>. (7 May 2011, date online accessed).
- [54] Svoboda, M. and Mlynkova, I. (2011) Correction of Invalid XML Documents with Respect to Single Type Tree Grammars. In Fong, S. (ed.), *Networked Digital Technologies*, Communications in Computer and Information Science 136, pp. 179–194. Springer, Berlin, Heidelberg.
- [55] Bouchou, B., Cheriati, A., Alves, M.H.F. and Savary, A. (2006) Integrating Correction into Incremental Validation. *22emes Journées Bases de Données Avancées, BDA (informal proceedings)*, Lille, France, October 17–20.
- [56] S. Staworko, J.C. (2006) Validity-Sensitive Querying of XML Databases. *Proc. EDBT 2006 Workshops on Current Trends in Database Technology*, Munich, Germany, March 26–31, Lecture Notes in Computer Science 4254, pp. 164–177. Springer, Berlin/Heidelberg.
- [57] Boobna, U. and de Rougemont, M. (2004) Correctors for XML Data. *Proc. EDBT 2004 Workshops on Current Trends in Database Technology*, Toronto, Canada, August 29–30, Lecture Notes in Computer Science 3186, pp. 69–96. Springer, Berlin/Heidelberg.
- [58] Flesca, S., Furfaro, F., Greco, S. and Zumpano, E. (2005) Querying and Repairing Inconsistent XML Data. *Proc. 6th Int. Conf. Web Information Systems Engineering*, New York, NY, USA, November 20–22, Lecture Notes in Computer Science 3806, pp. 175–188. Springer, Berlin/Heidelberg.
- [59] Tan, Z., Zhang, Z., Wang, W. and Shi, B. (2007) Computing Repairs for Inconsistent XML Document using Chase. *Proc. Int. Conf. Advances in Data and Web Management*, Huang Shan, China, June 16–18, Lecture Notes in Computer Science 4505, pp. 293–304. Springer, Berlin/Heidelberg.
- [60] Corrector prototype implementation. <http://www.ksi.mff.cuni.cz/~svoboda/>. (7 May 2011, date online accessed).
- [61] Levenshtein, V.I. (1966) Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, **10**, 707–710.
- [62] SAX Project. <http://www.saxproject.org/>. (7 May 2011, date online accessed).
- [63] Choi, B. (2002) What are Real DTDs Like? *Proc. 5th Int. Workshop on the Web and Databases*, Madison, WI, USA, June 6–7, pp. 43–48. ACM, New York, NY, USA.
- [64] McDowell, A., Schmidt, C. and Yue, K. (2004) Analysis and Metrics of XML Schema. *Proc. Int. Conf. Software Engineering Research and Practice*, Las Vegas, NV, USA, July 12–15, pp. 538–544. CSREA Press, Las Vegas, NV, USA.
- [65] Hosoya, H. and Pierce, B.C. (2003) XDuce: a statically typed XML processing language. *ACM Trans. Internet Technol.*, **3**, 117–148.
- [66] Onder, R. and Bayram, Z. (2006) XSLT version 2.0 is Turing-complete: A Purely Transformation Based Proof. In Ibarra, O.H. and Yen, H.-C. (eds), *Proc. 11th Int. Conf. Implementation and Application of Automata*, Taipei, Taiwan, August 21–23, Lecture Notes in Computer Science 4094, pp. 275–276. Springer, Berlin/Heidelberg.
- [67] Bruno, N., Koudas, N. and Srivastava, D. (2002) Holistic Twig Joins: Optimal XML Pattern Matching. *Proc. 2002 ACM SIGMOD Int. Conf. Management of Data*, Madison, WI, USA, June 3–6, pp. 310–321. ACM, New York, NY, USA.
- [68] ISO (1996) *ISO/IEC 14977:1996(E), Information technology—Syntactic metalanguage—Extended BNF*.
- [69] Draper, D., Fankhauser, P., Fernández, M., Malhotra, A., Rose, K., Rys, M., Siméon, J. and Wadler, P. (2010) *XQuery 1.0 and XPath 2.0 Formal Semantics*. W3C. <http://www.w3.org/TR/xquery-semantics/>. (7 May 2011, date online accessed).
- [70] Zhang, X., Pielech, B. and Rundessteiner, E.A. (2002) Honey, I Shrunk the XQuery!: An XML Algebra Optimization Approach. *Proc. 4th Int. Workshop on Web Information and Data Management*, McLean, VA, USA, November 4–9, pp. 15–22. ACM, New York, NY, USA.
- [71] Fokoue, A., Rose, K., Siméon, J. and Villard, L. (2005) Compiling XSLT 2.0 into XQuery 1.0. In Ellis, A. and Hagino, T. (eds), *Proc. 14th Int. Conf. World Wide Web*, Chiba, Japan, May 10–14, pp. 682–691. ACM, New York, NY, USA.
- [72] Kay, M. (2007) *XSL Transformations (XSLT) Version 2.0*. <http://www.w3.org/TR/xslt20/>. (7 May 2011, date online accessed).
- [73] Merlo, E., Kontogiannis, K. and Girard, J. (1992) Structural and Behavioral Code Representation for Program Understanding. *Proc. 5th Int. Workshop on Computer-Aided Software Engineering*, Montreal, QC, Canada, July 6–10, pp. 106–108. IEEE Computer Society, Washington, DC, USA.
- [74] Afanasiev, L. and Marx, M. (2008) An analysis of XQuery benchmarks. *Inform. Syst.*, **33**, 155–181.
- [75] Chamberlin, D., Fankhauser, P., Florescu, D., Marchiori, M. and Robie, J. (2007) *XML Query Use Cases*. W3C. <http://www.w3.org/TR/xquery-use-cases/>. (7 May 2011, date online accessed).
- [76] W3C (2006) *XML Query Test Suite*. <http://dev.w3.org/2006/xquery-test-suite/PublicPagesStagingArea/>. (7 May 2011, date online accessed).
- [77] Klettke, M., Schneider, L. and Heuer, A. (2002) Metrics for XML Document Collections. *Proc. Workshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, Prague, Czech Republic, March 25–27, pp. 15–28. Springer, London, UK.
- [78] Barbosa, D., Mignet, L. and Veltri, P. (2006) Studying the XML Web: Gathering statistics from an XML sample. *World Wide Web*, **9**, 187–212.
- [79] Kosek, J., Kratký, M. and Snášel, V. (2003) Struktura Reálných XML Dokumentu a Metody Indexovani. *Proc. 2003 Workshop on Information Technologies—Applications and Theory (Informal Proceedings)*, High Tatras, Slovakia, September 21. Czech.
- [80] W3C (August 2002) *The Extensible HyperText Markup Language (2nd edn)*. W3C. <http://www.w3.org/TR/xhtml1/>. (7 May 2011, date online accessed).
- [81] Bourret, R. (2005) *XML and databases*. <http://www.rpbouret.com/xml/XMLAndDatabases.htm>. (7 May 2011, date online accessed).
- [82] DeRose, S., Daniel, R., Grosso, P., Maler, E., Marsh, J. and Walsh, N. (2002) *XML Pointer Language (XPointer)*. W3C. <http://www.w3.org/TR/xptr/>. (7 May 2011, date online accessed).
- [83] DeRose, S., Maler, E. and Orchard, D. (2001) *XML Linking Language (XLink) Version 1.0*. W3C. <http://www.w3.org/TR/xlink/>. (7 May 2011, date online accessed).

- [84] Gao, S., Sperberg-McQueen, C.M. and Thompson, H.S. (2009) *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. W3C. <http://www.w3.org/TR/xmlschema11-1/>. (7 May 2011, date online accessed).
- [85] Peterson, D., Biron, P.V., Malhotra, A. and Sperberg-McQueen, C.M. (2009) *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. W3C. <http://www.w3.org/TR/xmlschema11-2/>. (7 May 2011, date online accessed).
- [86] Jelliffe, R. (2001) *The Schematron—an XML structure validation language using patterns in trees*. <http://xml.ascc.net/resource/schematron/>. (7 May 2011, date online accessed).
- [87] W3C (since 1994) *The Semantic Web Homepage*. W3C. www.w3.org/2001/sw/. (7 May 2011, date online accessed).
- [88] Smith, M.K., Welty, C. and McGuinness, D.L. (2004) *OWL Web Ontology Language*. <http://www.w3.org/TR/owl-guide/>. (7 May 2011, date online accessed).

Chapter 7

Conclusion

We have presented selected results of author's research, carried out through years 2007–2012 in cooperation with members of the *XML and Web Engineering Research Group*. Its core lies in proposal and implementation of a five-level evolution management framework which enables to design and maintain XML data structures which are processed within a complex application. In Chapter 2 we have described the architecture of the framework and focussed on its PIM and PSM levels (first in general, then from the XML perspective), their mutual mapping, respective operations and their propagation. In Chapter 3 we have moved to lower schema and intensional level and we have dealt with an approach for adapting XML documents with regard to changes in their XML schema. The output of the approach is an XSLT script which enables to re-validate the XML documents.

The following two sections have focussed on the process of reverse engineering, i.e., integration of an existing XML schema to the framework. First, in Chapter 4 we have described a set of improvements of a classical heuristic approach to XML schema inference for a sample set of XML documents, using exploitation of other input information, optimization of the inference process, or inferring new constructs. Hence, in Chapter 5 we can assume that we have an XML schema and we deal with structural and semantic similarity of XML schemas (expressed either in DTD or XML Schema). The similarity evaluation strategy is the core algorithm exploited further in the process of mapping of PSM of an XML schema to an existing PIM.

Last but not least, in Chapter 6 we introduce a “side” result of our research that has been exploited during several previous steps – an extensible framework for statistical analyses of real-world data. Its primary aim was

an analysis of real-world XML data and XML operations (queries); however, the tool is general and using plug-ins it can be extended for any kind of data.

7.1 Current and Future Research

The provided description may indicate, that the evolution management framework is finished and all the related problems are solved. As we have mentioned, papers forming Chapters 2 and 3 provide the common core of the proposal which can be further extended, generalized and optimized. In XRG we currently focus on several such improvements and extensions. Here we mention only a few of them, especially those on which the author of this thesis cooperates.

From the point of view of Figure 1.3 on page 7 in this thesis we have described the blue part of the framework – XML view – except for the operational level. However, as we have indicated in the introduction, a change in a particular data structure can affect not only other data structures, but also related operations, storage strategies, business-process models or, in general, any kind of related data model and/or operation. Hence, currently we are dealing with extension of the basic idea towards a more general tool.

Regarding the missing part of XML view – operational level – we have recently proposed a preliminary approach [39] which deals with the problem of adaptation of XML queries, in particular a subset of XPath (denoted in Figure 1.5 (d) on page 11 with the red color). This approach is a part of a new, more general implementation of the five-level evolution management framework, called *DaemonX* [13], which does not cover only the XML view, but in general any kind of view. It is implemented as a general framework, where new plug-ins can be added for new types of data. Currently it supports XML modeling, ER modeling and business-process modeling and we focus on its extending towards the full expected functionality. For instance, in [12] the framework is extended with a basic approach for adaptation of relational schemas, whereas in [23] change propagation to business process models is solved. On the other hand, in [38] the system is extended with integrity constraints, while in [18] efficient management of undo and redo operations in the environment of multiple schemas of different type is designed and implemented.

Another important current and future research direction covers the problem of inference of XML schemas. Also in this case we have recently provided

an implementation of a general framework called *jInfer* [20]. It supports the general inference process described in Section 1.3; however, using plug-ins it can be modified and extended so that different types of steps of the process can be applied and compared. So far we have extended the framework towards various optimization strategies of the inference process [21], exploitation of the knowledge of XML operations [27] or inference of integrity constraints [43, 44].

Our general aim in all the cases is still the same – to provide robust tools with real-world motivation, rich formal background, and user-friendly interface which can be easily extended with new, more efficient techniques.

Bibliography

- [1] *Service Oriented Architecture – SOA*. IBM. <http://www-01.ibm.com/software/solutions/soa/>.
- [2] *HTML 4.01 Specification*. W3C, 1999. <http://www.w3.org/TR/html4/>.
- [3] *ISO/IEC 9075-14:2003 Part 14: XML-Related Specifications (SQL/XML)*. ISO, 2006.
- [4] *SOAP Version 1.2 Part 0: Primer*. W3C, 2007. <http://www.w3.org/TR/soap12-part0/>.
- [5] *Web Services Business Process Execution Language (WSBPEL) TC*. OASIS, 2007. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.
- [6] *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*. W3C, 2007. <http://www.w3.org/TR/wsdl20-primer/>.
- [7] *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C, 2008. <http://www.w3.org/TR/REC-xml>.
- [8] *Web Services Activity*. W3C, 2009. <http://www.w3.org/2002/ws/>.
- [9] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simon. *XQuery 1.0: An XML Query Language*. W3C, 2007.
- [10] A. Boronat, J. A. Carsí, and I. Ramos. Algebraic Specification of a Model Transformation Engine. In *FASE '06: Proc. of the 9th Int. Conf. Fundamental Approaches to Software Engineering, Vienna, Austria*, volume 3922 of *LNCS*, pages 262–277. Springer, 2006.

- [11] P.P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
- [12] M. Chytil. *Adaptation of Relational Database Schema*. Master Thesis, Charles University in Prague, Czech Republic, 2012. <http://www.ksi.mff.cuni.cz/~mlynkova/dp/Polak.pdf>.
- [13] M. Chytil, K. Jakubec, V. Kudelas, P. Piják, M. Polák, M. Nečaský, and I. Mlýnková. DaemonX – Design Adaptation Evolution and Management of Native XML, 2011. <http://daemonx.codeplex.com/>.
- [14] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing Dependent Changes in Coupled Evolution. In *Proc. of the 2nd Int. Conf. on Model Transformations, ICMT 2009, Zurich, Switzerland*, volume 5563 of *LNCS*, pages 35–51. Springer, 2009.
- [15] K. Czarnecki and S. Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Syst. J.*, 45(3):621–645, 2006.
- [16] E. M. Gold. Language Identification in the Limit. *Information and Control*, 10(5):447–474, 1967.
- [17] ISO/IEC 9075-1:2008. *Part 1: Framework (SQL/Framework)*. ISO, 2008.
- [18] K. Jakubec. *Management of Undo/Redo Operations in Complex Environments*. Master Thesis, Charles University in Prague, Czech Republic, 2012. <http://www.ksi.mff.cuni.cz/~mlynkova/dp/Jakubec.pdf>.
- [19] M. Kay. *XSL Transformations (XSLT) Version 2.0*. W3C, 2007. <http://www.w3.org/TR/xslt20/>.
- [20] M. Klempa, M. Mikula, R. Smetana, M. Švirec, and M. Vitásek. *jInfer XML Schema Inference Framework*. <http://jinfer.sourceforge.net/>.
- [21] M. Klempa, J. Stárka, and I. Mlýnková. Optimization and refinement of xml schema inference approaches. *Procedia Computer Science*, 10(0):120–127, 2012.

- [22] J. Klímek, I. Mlýnková, and M. Nečaský. A Framework for XML Schema Integration via Conceptual Model. In *Web Information Systems Engineering WISE 2010 Workshops, WISS'10*, pages 84–97, Berlin, Heidelberg, 2011. Springer-Verlag.
- [23] V. Kudelas. *Adapting Service Interfaces when Business Processes Evolve*. Master Thesis, Charles University in Prague, Czech Republic, 2012.
- [24] J. Malý, M. Nečaský, and I. Mlýnková. Efficient Adaptation of XML Data Using a Conceptual Model. *Information Systems Frontiers*, 2012. (in press).
- [25] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, 2006.
- [26] L. Mignet, D. Barbosa, and P. Veltri. The XML Web: a First Study. In *WWW '03: Proc. of the 12th Int. Conf. on World Wide Web, Volume 2*, pages 500–510, New York, NY, USA, 2003. ACM Press.
- [27] M. Mikula, J. Stárka, and I. Mlýnková. Inference of an XML Schema with the Knowledge of XML Operations. In *SITIS '12: Proceedings of the 8th International Conference on Signal-Image Technology and Internet-Based Systems*, Naples, Italy, 2012. IEEE Computer Society Press. (in press).
- [28] J. Miller and J. Mukerji. *MDA Guide Version 1.0.1*. OMG, 2003. <http://www.omg.org/docs/omg/03-06-01.pdf>.
- [29] I. Mlýnková. An Analysis of Approaches to XML Schema Inference. In *Int. IEEE Conf. on Signal-Image Technologies and Internet-Based System*, pages 16–23, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
- [30] I. Mlýnková and M. Nečaský. Heuristic Methods for Inference of XML Schemas: Lessons Learned and Open Issues. *Informatika – an International Journal*, 2012. (in press).
- [31] I. Mlýnková, K. Toman, and J. Pokorný. Statistical Analysis of Real XML Data Collections. In *COMAD '06: Proc. of the 13th Int. Conf.*

on *Management of Data*, pages 20–31, New Delhi, India, 2006. Tata McGraw-Hill Publishing Company Limited.

- [32] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML Schema Languages using Formal Language Theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.
- [33] M. Nečaský. *Conceptual Modeling for XML*, volume 99 of *Dissertations in Database and Information Systems*. IOS Press, Amsterdam, Netherlands, 2009.
- [34] M. Nečaský, J. Klímek, J. Malý, and I. Mlýnková. Evolution and Change Management of XML-based Systems. *Journal of Systems and Software*, 85(3):683–707, 2012.
- [35] OMG. *MOF QVT Final Adopted Specification*. Object Modeling Group, June 2005. http://fparreiras/papers/mof_qvt_final.pdf.
- [36] OMG. *UML Infrastructure Specification 2.1.2*, nov 2007. <http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/>.
- [37] OMG. *UML Superstructure Specification 2.1.2*, nov 2007. <http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/>.
- [38] V. Piják. *Universal Constraint Language*. Master Thesis, Charles University in Prague, Czech Republic, 2011.
- [39] M. Polák. *XML Query Adaptation*. Master Thesis, Charles University in Prague, Czech Republic, 2011. <http://www.ksi.mff.cuni.cz/~mlynkova/dp/Polak.pdf>.
- [40] J. Stárka, I. Mlýnková, J. Klímek, and M. Nečaský. Integration of Web Service Interfaces via Decision Trees. In *Proc. of the 7th Int. Symposium on Innovations in Information Technology*, pages 47–52, Washington, DC, USA, 2011. IEEE Computer Society.
- [41] J. Stárka, M. Svoboda, J. Sochna, J. Schejbal, I. Mlýnková, and D. Bednárek. *Analyzer: A Complex System for Data Analysis*. *Comput. J.*, 55(5):590–615, 2012.

- [42] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures (Second Edition)*. W3C, October 2004. <http://www.w3.org/TR/xmlschema-1/>.
- [43] M. Vitásek and I. Mlýnková. Inference of XML Integrity Constraints. In *Advances in Databases and Information Systems*, volume 186 of *Advances in Intelligent Systems and Computing*, pages 285–296. Springer Berlin Heidelberg, 2013.
- [44] M. Švirec and I. Mlýnková. Efficient Detection of XML Integrity Constraints Violation. In *Networked Digital Technologies*, volume 293 of *Communications in Computer and Information Science*, pages 259–273. Springer Berlin Heidelberg, 2012.
- [45] A. Wojnar, I. Mlýnková, and J. Dokulil. Structural and Semantic Aspects of Similarity of Document Type Definitions and XML Schemas. *Information Sciences*, 180(10):1817–1836, 2010. Special Issue on Intelligent Distributed Information Systems.