eXolutio: Tool for XML Schema and Data Management*

Jakub Klímek, Jakub Malý, Irena Mlýnková, and Martin Nečaský

XML and Web Engineering Research Group, Department of Software Engineering
Faculty of Mathematics and Physics, Charles University in Prague
Malostranské náměstí 25, 118 00 Praha 1, The Czech Republic
{klimek, maly, mlynkova, necasky}@ksi.mff.cuni.cz

Abstract. Recently XML has achieved the leading role among languages for data representation and, thus, the amount of related technologies and applications exploiting them grows fast. However, only a small percentage of applications is static and remains unchanged since its first deployment. Most of the applications change with newly coming user requirements and changing environment. In this paper we describe a tool for evolution and change propagation of XML applications called *eXolutio*, which has been developed and improved in our research group during last few years. The text should help the reader to get acquainted with the tool and its theoretical background.

Keywords: XML schema, conceptual modeling, tool, evolution

1 Introduction

The eXtensible Markup Language (XML) is currently a de-facto standard for data representation and together with accompanying standards, such as XML Schema, XPath, XQuery, XSLT, 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 syntax of the XML documents in a form of *XML schemas* expressed in some XML schema language, e.g. DTD or XML Schema. 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 represent particular views of the application domain of the software system. We can, therefore, speak about a *family of XML formats* utilized 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. Each such change may influence many different XML formats in the family. Without a proper technique, we have to identify the XML formats affected by the change manually and ensure that they are evolved coherently with each

* This work was supported in part by the Czech Science Foundation (GAČR), grant numbers P202/10/0573 and P202/11/P455 and in part by the grant SVV-2012-265312.

J. Pokorný, V. Snášel, K. Richta (Eds.): Dateso 2012, pp. 69–80, ISBN 978-80-7378-171-2.

other. Such evolution brings a challenge for research, so that the user interaction and, hence, the expensive and error-prone work can be minimized. We cannot leave the user out completely, there still remain cases when user decision is unavoidable; however, automatic management of evolution enables to identify all the affected parts of the application and perform the user-selected changes correctly and efficiently and, possibly, exploit them in further automatic processing.

In our research group we have focused on the area of efficient and correct management of a family of XML formats for several recent years. Starting with a simple idea of propagation of changes among related XML formats, we have gradually extended our effort towards a robust framework and its implementation in a tool called *eXolutio*. It currently supports the original idea of designing XML formats using the principles of *Model Driven Architecture* (MDA) [18], their evolution, and integration of new XML formats into the framework.

Contributions The aim of this paper is to provide a covering overview of our research in the area of XML evolution, to describe architecture and implementation of the *eXolutio* tool, to present results of our experiments with the tool proving the concept and efficiency and a comparison of the tool with similar tools.

Outline The rest of the paper is structured as follows: In Section 2 we focus on the background theoretical aspects – our conceptual model for XML. In Section 3 we introduce *eXolutio*, our tool in which we implement our research results. In Section 4 we provide the proof of the concept using a set of experiments. In Section 5 we discuss the related work. Finally, in Section 6 we conclude.

2 Conceptual Model for XML

In this section, we introduce our conceptual model for XML and its inheritance extension. It has two levels, PIM and PSM, which is inspired by MDA.

A PIM schema is based on UML class diagrams and models real-world concepts and relationships among them. It contains three types of components: classes, attributes and associations. A sample PIM schema is depicted in Figure 1. Where association cardinality is not explicitly stated, default cardinality `1..1` applies. A part of the PIM are also integrity constraints which we are currently working on, but which are not in the scope of this paper.

Definition 1. *A* platform-independent (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.*

- Class $C \in \mathcal{S}_c$ *has a name assigned by function* name*. For inheritance purposes, function* isa *assigns a parent class to a child class (UML generalization) and must not form a cycle. Furthermore, functions* abstract *and* final *determine whether the class can have instances in data and whether this class can be inherited from, respectively.*
- 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*.*

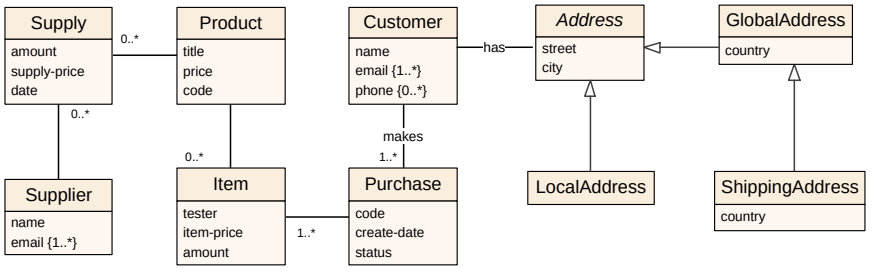


Fig. 1. Example of a PIM diagram

– 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$(R) = \lambda$*.*

For a class $C \in \mathcal{S}_c$*, we will use* attributes(C) *to denote the set of all attributes of* C*, i.e.* attributes$(C) = \{A \in \mathcal{S}_a : \text{class}(A) = C\}$*. Similarly,* associations(C) *will denote the set of all associations with* C *as a participant, i.e.* associations$(C) = \{R \in \mathcal{S}_r : (\exists E \in R)(\text{participant}(E) = C)\}$*. For a given association* $R = (E_1, E_2)$*, we will use notation* (C_1, C_2) *as an equivalent of* $(participant(E_1), participant(E_2))$ *if there are no more associations connecting* C_1 *and* C_2*.*

The *platform-specific model (PSM)* specifies how a part of the reality is represented in a particular XML schema in a UML-style way. We introduce it formally in Definition 2. We view a PSM schema in two perspectives. From the *grammatical perspective*, it models XML elements and attributes. From the *conceptual perspective*, it delimits the represented part of the reality. Its advantage is that the designer works in a UML-style way which is more comfortable then editing the XML schema. Formally, there is a mapping from each PSM schema to the PIM schema.

Definition 2. *A* platform-specific (PSM) schema *is a 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*.*

– Class $C' \in \mathcal{S}'_c$ *has a name assigned by function* name*. For inheritance purposes, function* isa *assigns a parent class to a child class and the relation must not form a cycle. Furthermore, functions* abstract *and* final *determine whether the class can have instances in data and whether this class can be inherited from, respectively.*
– Attribute $A' \in \mathcal{S}'_a$ *has a name, data type, cardinality and XML form (whether it models an XML attribute or an XML element) assigned by functions* name*,* type*,* card *and* xform*, respectively.* xform$(A') \in \{\text{e}, \text{a}\}$*. 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')*.*

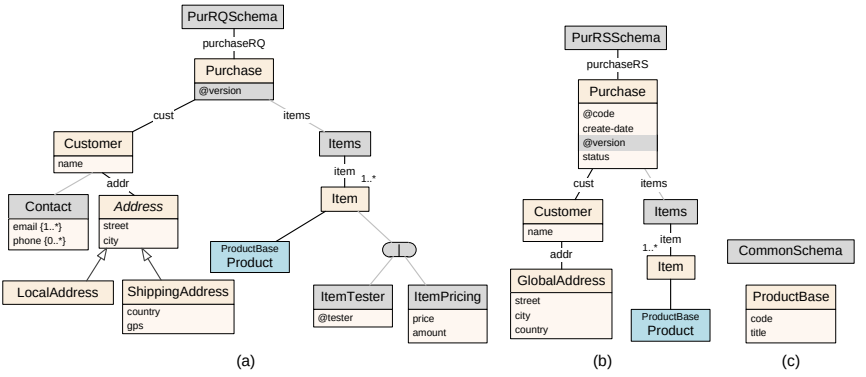


Fig. 2. Examples of PSM schemas

- 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* R'*. Both* E'_1 *and* E'_2 *have a cardinality assigned by function* card *and each is associated with a class from* $\mathcal{S}'_c$ *or content model from* $\mathcal{S}'_m$ *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$(R') = \lambda$.
- Content model $M' \in \mathcal{S}'_m$ *has a content model type assigned by function* cmtype. cmtype$(M') \in \{\texttt{sequence}, \texttt{choice}, \texttt{set}\}$.

The graph $(\mathcal{S}'_c \cup \mathcal{S}'_m, \mathcal{S}'_r)$ *must be a forest*[1] *of rooted trees with one of its trees rooted in* $\mathcal{C}'_{\mathcal{S}'}$*. For* $C' \in \mathcal{S}'_c$*, attributes* (C') *will denote the sequence of all attributes of* C' *ordered by* position*, i.e.* attributes $(C') = (A'_i \in \mathcal{S}'_a : \text{class}(A'_i) = C' \wedge i = \text{position}(A'_i))$*. Similarly,* content (C') *will denote the sequence of all associations with* C' *as a parent ordered by* position*, i.e.* content $(C') = (R'_i \in \mathcal{S}'_r : \text{parent}(R'_i) = C' \wedge i = \text{position}(R'_i))$*. We will call* content (C') content of C'.

A sample PSM schema is depicted in Figure 2. PSM uses similar constructs to PIM: classes, attributes and associations. The PSM-specific constructs have precisely defined semantics. A class models a complex content. The complex content is specified by the attributes of the class and associations in its content (their ordering is given by functions *attributes* and *content*). An attribute models an XML element declaration with a simple content or XML attribute declaration depending on its XML form (function *xform*). An association models an XML element declaration with a complex content if it has a name. Otherwise, it models only that the complex content modeled by its child is nested in the complex content modeled by its parent. Type of the modeled content (set,

[1] Note that since $\mathcal{S}'$ 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.

choice, sequence) can be specified by a special construct that can be, for example, seen in Figure 2(a) under the `Item` class.

2.1 Interpretation of PSM schema against PIM schema

A PSM schema represents a part of a PIM schema. A class, attribute or association in the PSM schema may be mapped to a class, attribute or association in the PIM schema. In other words, there is a mapping which specifies the semantics of classes, attributes and associations of the PSM schema in terms of the PIM schema. The mapping must meet certain conditions to ensure consistency between PIM schemas and the specified semantics of the PSM schema. The interpretation of a PSM schema against a PIM schema is what we call the mapping. It is the core feature of our conceptual model. It interconnects constructs on the platform-specific level with those on the platform-independent level and allows for interesting use cases for the conceptual model like XML schema evolution and integration [13, 14, 19, 20]. Its definition is, however, not trivial and is beyond the scope of this paper.

3 eXolutio architecture

The implementation of our research results is a tool called *eXolutio* [12]. There exists also an older version of our conceptual model and its implementation called *XCase* [11], which is the predecessor of *eXolutio*. For simplicity, we will stick to the current name. *eXolutio* allows the user to model a PIM schema and multiple PSM schemas with interpretations against the PIM schema. The user can then evolve the whole set of schemas coherently, because his operations are propagated to all affected places by a mechanism described in [19].

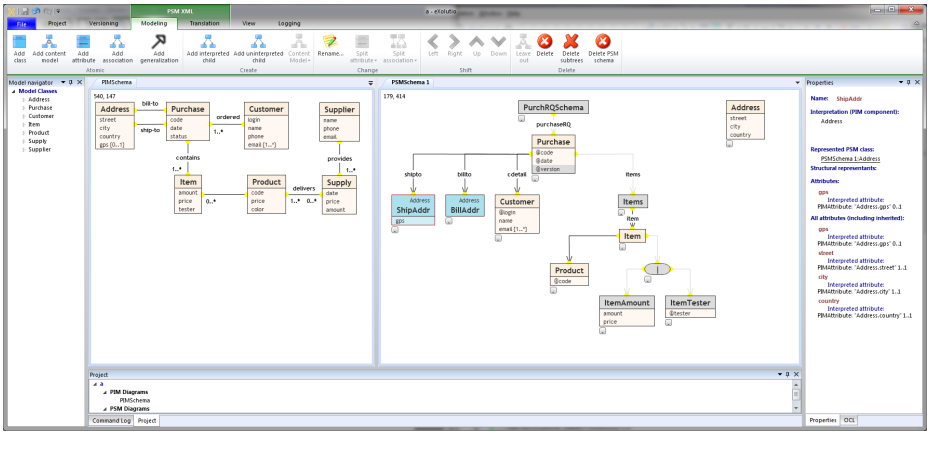
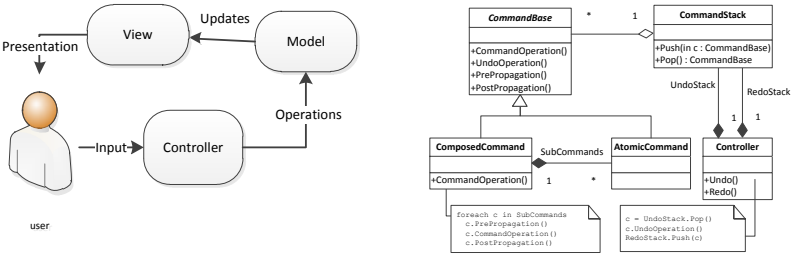


Fig. 3. *eXolutio* screenshot



(a) Overall architecture (b) Controller

Fig. 4. eXolutio – main MVC components

The architecture of *eXolutio* is based on the Model–View–Controller (MVC) design pattern (Figure 4(a)). This means that we hold all the project data in the model part, neither mixing it with operations, nor visualization. Whenever a user issues a command, it is handled by the controller part. The controller makes all the necessary changes in the model. The view part observes that the model has changed and updates the visualization. The connections between individual parts are loose enough so it is possible to, e.g., use multiple visualizations. In particular, we have a Windows Presentation Foundation [17] visualization (a desktop application) a Silverlight [16] visualization (a web application) and a no-visualization (a console application) versions of *eXolutio* which all share the same model and the same controller.

Model The model part of the tool based on our conceptual model [21] consists of classes for each modeled component, such as a class, an association or an attribute on each of the modeled levels (PIM and PSM), a class for PIM and PSM schemas and a class for the whole project. Besides the obvious properties of components like a name or a collection of attributes of a class, each component class implements methods for serialization and deserialization of the component to and from XML. Therefore, when we save and load a project, we simply call a serialization or a deserialization method on all found objects in a certain order. Finally, each schema contains lists of all the components of individual types in that schema, so we can easily go through, e.g., all associations in a certain schema. Since one of the main features of our tool is the visualization of connections between the two levels of abstraction, one of the most common queries is “Give me all PSM classes which have this PIM class as their interpretation”. We basically go through each PSM schema in the project and through each PSM class in that schema and check whether its interpretation is the given PIM class. In addition, the model contains methods for easy traversal of both the PIM and PSM schemas. An example can be a method for retrieval of all attributes of a PSM class including those inherited by the structural representative constructs. Another example can be a method that gets all uninterpreted descendants of an interpreted PSM class. When a certain method representing a query over the model is needed by the controller more than once, we make it a model method so that everyone can use it.

View The view component serves for two purposes: it visualizes the model for the user and provides user-friendly interface to run the controller commands. PIM schemas are depicted as UML diagrams and the layout of the diagram is left up to the user preference, for PSM schemas we use automatic hierarchical layouting to emphasize the fact that a PSM schema is a tree/forest. Besides the visualizations of the schemas, view component provides several windows and controls that help the user to navigate the modeled project, see the connections between individual concepts and follow the various links (e.g. find interpretation of an attribute or a class referred from a structural representant). The view component can be run either as a desktop application or inside a web browser using Silverlight plugin technology. This browser view can be used to accompany a documentation of published XML schema standards (e.g. [1]). An interactive visualization of a family of schemas joined by a common model can benefit greatly every system designer, who wants to adopt a third party standard and needs a clear overview of the whole problem domain and its individual schemas.

Controller The controller is the core of the tool. It contains all the operations and algorithms that make the tool unique. Also, it contains the usual command and undo/redo management. Whenever a user issues a command from view, it is handled by the controller. The controller (depicted in Figure 4(b)) gets all the necessary parameters from view such as what command is requested, the currently selected components, the new name for a component, etc. The controller creates the appropriate command, which in most cases will be one of our composite operations (described later in this section) and passes all the required parameters. The operation executes and updates the model accordingly. Then it places the command on the undo stack. The command itself contains all the information it needs to change the model back to the state it was in before the command was executed. In other words, we can simply call undo and the command knows what it needs to do and whether it is possible. This way, we can stack the executed commands and perform undo and redo operations as needed and as usual. In [19] we have described a theoretical background for atomic (simple, well defined) and composite (user-friendly) operations, which we will now describe from the implementation point of view. One of our goals was also to make the two levels of abstraction (PIM and PSM) work as independently of each other as possible while maintaining consistency when there are connections between the levels. Therefore, the operations need to work at their respective levels and be propagated only when there is an interpretation. Since we have a quite complex system of operations, we had to break it down into simpler parts. This means that among our atomic operations one can find for example an operation that creates an attribute. But it does not do anything else than that. Specifically, it does not give a name to the attribute, it does not set its datatype, etc. For that, we have other atomic operations. Having the atomic operations, we can compose more complex and user-friendly ones. A basic composite operation can be the already mentioned creation of an attribute, which this time is user friendly. It is composed of the creation of the attribute, renaming the attribute, setting its cardinality and its datatype. If it was a PSM attribute, the operation would also set its *xform* (Definition 2). So this is basically a predefined sequence of 4 or 5 operations, which is quite simple. Another simple example can be deletion of an attribute. This means setting its cardinality, name and datatype to default values and then deleting it. The reason for this is that when we undo

this operation, we want the name and the other values of the attribute to recover, so it is not correct to just delete the attribute. Let us have a look at a more advanced example.

So far we have described how to compose atomic and composite operations. However, these worked on their respective levels of abstraction. Now we have to make sure that when there is an interpretation of a PSM schema against a PIM schema, we keep the model in a consistent state and save the user's time by propagating the changes between the levels. This is achieved by the propagation. Before each atomic operation is executed, a method implementing its propagation to the other level is called. It determines whether there is an interpretation and therefore the need to propagate. If so, it creates a (possibly) composite operation on the other level of abstraction and integrates it to the currently running operation. Only when the propagation succeeds, the original atomic operation that caused it is executed. This way, the propagation actually becomes a part of the currently running operation. This is convenient because when it finishes, it can be undone and redone like any other operation.

4 Experiments

To provide the proof of the whole concept and show the advantages of our tool, we evaluate our approaches using experiments based on a real-world family of XML schemas. We experiment with the *National Register for Public Procurement System* (NRPP)[2]. It is a governmental information system where public authorities in the Czech Republic publish data about their public contracts. Authorities send contract information to the information system formatted in one of the 17 XML formats accepted by the NRPP. This includes, e.g. XML format for contract notifications, supplier selection notifications, etc. The information is then published by the system in the form of HTML pages. The goal of the experiment is to show how our approach would save time if the authors of the NRPP XML formats used *eXolutio* to design the XML formats and evolve them according to changing legislation instead of their manual editing and adaptation.

Currently, the NRPP only provides a textual documentation for the XML formats and a set of sample XML documents. Therefore, our first goal is to design a conceptual schema in a form of a PIM schema which models the domain of public contracts and design PSM schemas of the XML formats mapped to the PIM schema.

The PIM schema contains classes which model public contracts and their procurers and suppliers. There are also some additional concepts modeled – i.e. prices and contact information. A supplier is associated with a contract, 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 and where bids to the contract are collected. Finally, there are four different prices – expected price, the best offered price, price agreed by a selected supplier and procurer, and a final real price known after finishing the contract. The PSM schema depicted in Figure 5 (a) models an XML format which a public authority uses to send a notification about a new public contract to NRPP. The PSM schema depicted in Figure 5 (b) models an XML format for notifications about the supplier selected for the contract.

[2] `http://www.isvz.cz` (in Czech only)

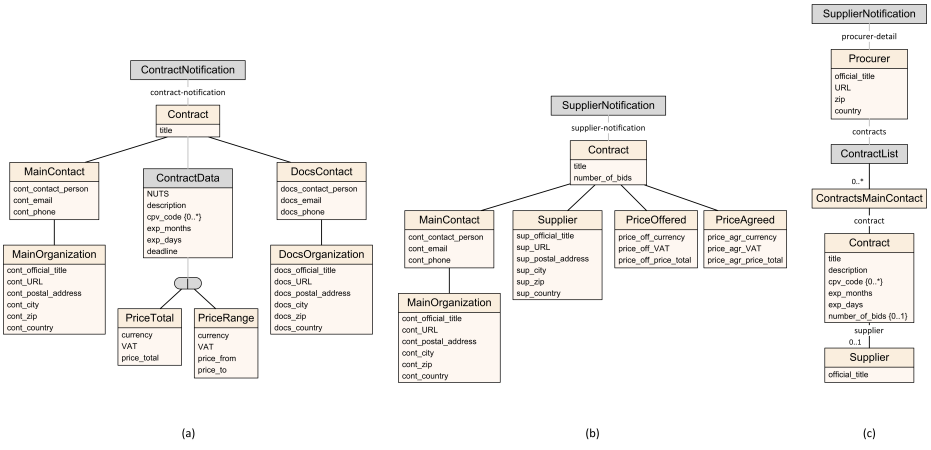


Fig. 5. PSM schemas modeling XML formats for (a) sending contract notifications to NRPP, (b) reporting on contract supplier selection to the NRPP, and (c) representing procurer detail

We can measure the amount of manual work required to design the PIM and PSM schemas in terms of numbers of executed atomic operations. The numbers of atomic operations executed to create the PIM and PSM schemas are depicted in Figure 6 (a). It shows that only creation and update operations were used. Here, manual creation of the schemas is necessary so there is no direct advantage in comparison to writing the XML schemas of the XML formats directly. However, *eXolutio* saves time because for each performed atomic operation it checks whether it does not break the consistency between the XML formats. When the designer codes the XML schemas of the XML formats directly no such control is performed automatically and (s)he must do it manually in each step during coding.

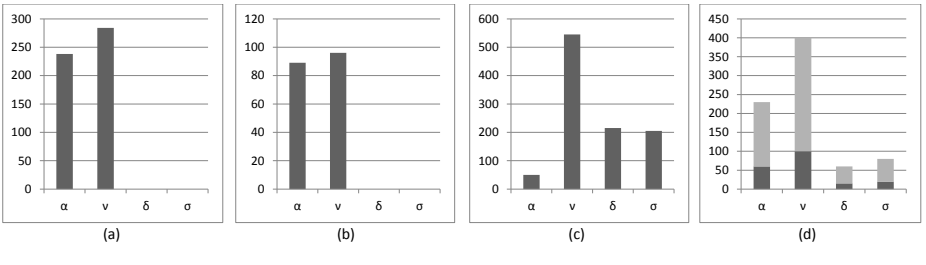


Fig. 6. Numbers of atomic operations performed manually by the designer (dark grey) and automatically by the propagation mechanism (light grey)

Having the PIM schema and a set of PSM schemas of the XML formats used by NRPP we set ourselves three goals. The first goal is to show how *eXolutio* facilitates creating new XML formats on the basis of an existing PIM schema. A PSM schema of

a new XML format for public procurer details is depicted in Figure 5 (c). The numbers of the atomic operations executed at this step are depicted in Figure 6 (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 shows 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 faster 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 second goal is to improve the quality of the NRPP XML formats, which 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. The numbers of the executed atomic operations are depicted in Figure 6 (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 when changes are performed.

The third goal was to show how the set of schemas can be evolved coherently. 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 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.

Finally, there was a requirement to update the XML format for contract notifications (Figure 5 (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. This change was correctly propagated to the PIM schema, because it is a conceptual change. From here, it was propagated to the other PSM schemas.

The numbers of the atomic operations executed during the last two steps are depicted in Figure 6 (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. α are additions, υ are changes, δ are deletions and σ are synchronizations - statements that two modeled sets of attributes or associations are semantically equivalent. The synchronizations are very useful in our change propagation mechanism, for details refer to [19, 20].

5 Related work

The current approaches towards evolution management can be classified according to distinct aspects [15, 8]. The changes and transformations can be expressed [22, 4] as

well as divided [6] variously too. However, to our knowledge there exists no general framework comparable to our proposal; particular cases and views of the problem have previously only been solved separately, superficially and mostly imprecisely without any theoretical or formal basis.

XML View We can divide the current approaches to XML schema evolution and change management into several groups. Approaches in the first group consider changes at the schema level and differ in the selected XML schema language, i.e. DTD [2, 7] or XML Schema [24, 5]. The changes are expressed variously and more or less formally. Approaches in the second and third group are similar, but they consider changes at an abstraction of logical level – either visualization [10] or a kind of UML diagram [9]. 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.

In all the papers cited the authors consider only a single XML schema. In [23] 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 [3].

6 Conclusion

In this paper, we introduced *eXolutio*, our tool for XML schema and data management. We surveyed related work and we showed the theoretical background behind our tool and evaluated it on real-world XML schemas.

References

1. *OpenTravel.org*.
2. L. Al-Jadir and F. El-Moukaddem. Once Upon a Time a DTD Evolved into Another DTD... In *Object-Oriented Information Systems*, pages 3–17, Berlin, Heidelberg, 2003. Springer.
3. Z. Bellahsene, A. Bonifati, and E. Rahm. *Schema Matching and Mapping*. Data-Centric Systems and Applications. Springer Berlin Heidelberg, 2011.
4. 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.
5. F. Cavalieri. EXup: an Engine for the Evolution of XML Schemas and Associated Documents. In *EDBT '10: Proc. of the 2010 EDBT/ICDT Workshops*, pages 1–10, New York, NY, USA, 2010. ACM.
6. A. Cicchetti, D. D. 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.
7. S. V. Coox. Axiomatization of the Evolution of XML Database Schema. *Program. Comput. Softw.*, 29(3):140–146, 2003.

8. K. Czarnecki and S. Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Syst. J.*, 45(3):621–645, 2006.
9. E. Domínguez, J. Lloret, A. L. Rubio, and M. A. Zapata. Evolving XML Schemas and Documents Using UML Class Diagrams. In *DEXA'05: Proc. of the 16th Int. Conf. on Database and Expert Systems Applications*, volume 3588 of *LNCS*, pages 343–352. Springer, 2005.
10. M. Klettke. Conceptual XML Schema Evolution – The CoDEX Approach for Design and Redesign. In M. Jarke, T. Seidl, C. Quix, D. Kensche, S. Conrad, E. Rahm, R. Klamma, H. Kosch, M. Granitzer, S. Apel, M. Rosenmüller, G. Saake, and O. Spinczyk, editors, *BTW'07*, pages 53–63, Aachen, Germany, March 2007.
11. J. Klímek, L. Kopenec, P. Loupal, and J. Malý. XCase - A Tool for Conceptual XML Data Modeling. In *Advances in Databases and Information Systems*, volume 5968/2010 of *Lecture Notes in Computer Science*, pages 96–103. Springer Berlin / Heidelberg, March 2010. `http://www.springerlink.com/content/v45198r1v783xu13`.
12. J. Klímek, J. Malý, and M. Nečaský. eXolutio – A Tool for XML Data Evolution, 2011. `http://exolutio.com`.
13. J. Klímek and M. Nečaský. Integration and Evolution of XML Data via Common Data Model. In *Proceedings of the 2010 EDBT/ICDT Workshops, Lausanne, Switzerland, March 22-26, 2010*, New York, NY, USA, 2010. ACM.
14. J. Klímek and M. Nečaský. Generating Lowering and Lifting Schema Mappings for Semantic Web Services. In *25th IEEE International Conference on Advanced Information Networking and Applications Workshops, WAINA 2010, Biopolis, Singapore, 22-25 March 2011*. IEEE Computer Society, 2011.
15. T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, 2006.
16. Microsoft. Silverlight. `http://www.microsoft.com/silverlight/`.
17. Microsoft. *Windows Presentation Foundation (WPF)*. December 2010. `http://msdn.microsoft.com/en-us/library/ms754130.aspx`.
18. J. Miller and J. Mukerji. *MDA Guide Version 1.0.1*. Object Management Group, 2003.
19. 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.
20. M. Nečaský, I. Mlýnková, and J. Klímek. Model-Driven Approach to XML Schema Evolution. In R. Meersman, T. S. Dillon, and P. Herrero, editors, *OTM Workshops*, volume 7046 of *Lecture Notes in Computer Science*, pages 514–523. Springer, 2011.
21. M. Nečaský, I. Mlýnková, J. Klímek, and J. Malý. When conceptual model meets grammar: A dual approach to XML data modeling. *Data & Knowledge Engineering*, 72(0):1 – 30, 2012.
22. OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.0*. Object Modeling Group, April 2008. `http://www.omg.org/spec/QVT/1.0/`.
23. K. Passi, D. Morgan, and S. Madria. Maintaining Integrated XML Schema. In *IDEAS '09: Proc. of the 2009 Int. Database Engineering, Applications Symp.*, pages 267–274, New York, NY, USA, 2009. ACM.
24. M. Tan and A. Goh. Keeping Pace with Evolving XML-Based Specifications. In *EDBT'04 Workshops*, pages 280–288, Berlin, Heidelberg, 2005. Springer.