Computing and Informatics, Vol. 34, 2015, 99–137

DATA AND QUERY ADAPTATION USING DAEMONX

Marek Polák, Martin Chytil, Karel Jakubec
Vladimír Kudelas, Peter Piják, Martin Nečaský
Irena Holubová (Mlýnková)

Department of Software Engineering
Charles University in Prague
Malostranské náměstí 25, 118 00 Praha 1, Czech Republic
e-mail: {polak, necasky, holubova}@ksi.mff.cuni.cz

Abstract. The most common applications of the today's IT world are information systems. The problems related to their design and implementation have sufficiently been solved. However, the true problems occur when an IS is already deployed and user requirements change. In this paper we introduce *DaemonX* – an evolution management framework which enables to manage evolution of complex applications efficiently and correctly. Using the idea of plug-ins, it enables to model almost any kind of a data format (currently XML, UML, ER, and BPMN). Since it preserves also mapping among modeled constructs of modeled formats via a common platform-independent model, it naturally supports propagation of changes to all related and affected parts.

Keywords: Evolution management, data adaptation, query adaptation

1 INTRODUCTION

The most common application of the today's information-technology (IT) world are *information systems* (IS). There exist various types of ISs, such as decision-support systems, database-management systems, etc. Currently a very popular kind of IS are distributed ISs. For instance, such an IS can be based on the *Service Oriented Architecture* (SOA) [1] and its most common implementation – *Web Services* (WS) [2].

The life-cycle of a complex system of applications is similar to the life-cycle of a single application; however, the complexity is much higher. We need to design nu-

merous 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 to all affected parts correctly and efficiently. The ability of an IS to adapt to the changes is called *adaptability*. Even though such ability is natural, currently it is being solved mostly manually by an IT expert who knows the IS well and is able to identify all its components that need to adapt. But, in a complex IS it is not possible for a single person to consider and cover all the components and aspects.

Another approach, utilized mainly by standardization entities is preservation of *backward compatibility*. For example, the open XML standard *OpenTravel.org*[1] currently offers 319 XML schemas which standardize communication in the traveling community. *OpenTravel.org* is changed twice a year and the changes are published in a form of a new version of the XML schemas and a documentation of the changes in a form of human-readable document. This solution, however, requires tremendous manual effort from designers to adapt their transformation scripts and potentially their database, their own XML formats and their program code as well.

To solve the indicated problem we need a robust evolution framework that enables to concurrently model all related parts of the system (i.e., data structures, ICS and operations) as precisely as possible (i.e., with a rich set of constructs), to preserve the relations between system components, and to enable respective change management (i.e., correct propagation of changes to all the affected parts). In this paper we introduce *DaemonX*[2] – an evolution management framework, which enables to manage evolution of complex applications efficiently and correctly on the basis of semi-automatic and formally well-founded strategies. Using the idea of plug-ins, it enables to model almost any kind of a data format (currently it supports XML, UML [3], ER [4], and BPMN [5]). Since it preserves relationships among the modeled constructs, it naturally supports also propagation of changes to all related affected parts. In this paper we first describe the proposal of *DaemonX* and its architecture in general and then we focus on several important and novel aspects of change management covered by the system. In particular, we show how such a general system enables to deal with change management of areas which are not covered in current literature much due to complexity, namely change propagation from XML schemas to XML queries expressed in XPath [6] and from relational schemas to SQL [7] queries.

The paper is structured as follows: In Section 2 we discuss the problem of evolution management in more detail and propose the general evolution framework.

---

[1] `http://www.opentravel.org`

[2] `http://daemonx.codeplex.com/`

In Section 3 we describe the architecture of *DaemonX*. The next two sections are devoted to selected representatives of successful evolution management using *DaemonX*. First, in Section 4 evolution of queries over evolving XML data is presented. In the following Section 5 evolution of relational model and SQL queries from the original paper is described. In Section 6 we describe the related papers relevant to our research. Finally, in Section 7 we conclude.

2 EVOLUTION MANAGEMENT

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 schemas. Consider the two sample XML documents in Figure 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 document, 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 a different representation than in the previous sample. We need an unified representation of shipping and billing addresses in purchase requests for all kinds of customers.

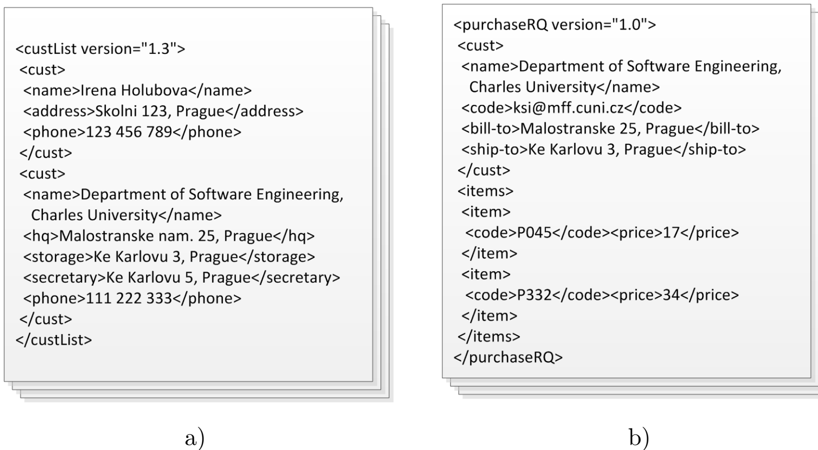

Figure 1. Sample XML documents from the same problem domain: a) XML format for list of customers, b) XML format for purchase requests

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. As indicated in the introduction, in a real-world application such a simple change can trigger a huge amount of necessary modifications. And, in general, the changes can be much more complex and ongoing process.

A natural and real-world solution of the evolution problem is to rely on an IT expert who is able to denote the part of the system which is modified and satisfies the following problems: (P1) *to make the required change easily and correctly*, (P2) *to identify all affected parts of the system*, that needs 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*.

To help to solve the indicated problems P1–P5, in our previous papers [8, 9] we have described the idea of a *five-level evolution management framework*. Using several levels of abstraction it enables one to model all parts of the system regardless technical details of a selected format. Preserved relations between the levels enable one to propagate the changes correctly among multiple related and overlapping schemas. In the first part of our research we focused on its first part – so-called *XML view*, i.e., evolution of a set of XML schemas. In this paper we extend the idea towards other possible formats, i.e., we consider the system in its full generality. The full architecture of the framework is depicted in Figure 2.

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 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 [10] or SQL/XML [11] 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).

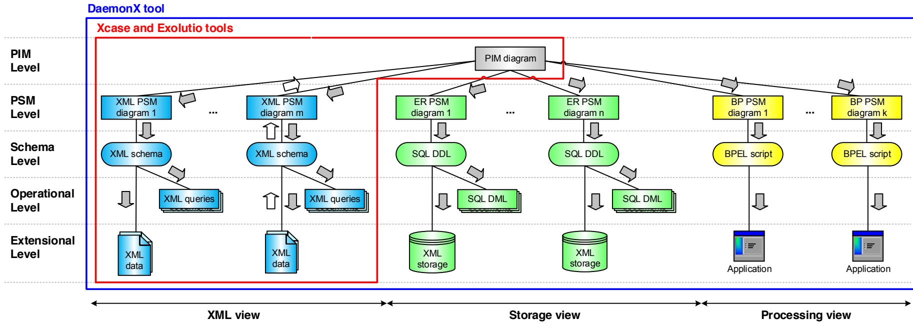

Figure 2. Five-level evolution management framework with depicted difference between *DaemonX* and our previous tools *XCase* and *eXolutio*

Considering only the three levels leads to evolution of each affected schema separately. However, this is a highly time-consuming and error-prone solution. Therefore, we introduce two additional levels, which follow the *model driven architecture* (MDA) [12] 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 [13] schemas which model hierarchical data structures implemented using a selected XML schema language or ER [14] 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 [15] or SQL DDL [7]) 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. In other words, instead of N mappings for each model we need only one. Hence, the change propagation is realized using this common point. For instance, if a change occurs in an XML schema, it is propagated to PSM schema and, finally, PIM schema. We speak about *upwards propagation*, in Figure 2 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 2 it is denoted by the grey arrows.

The described ideas have been first implemented as a project called *XCase*[3] and later re-engineered into a more general system called *eXolutio*[4]. *DaemonX* is another step in our effort. On the basis of the lessons learned from the two previous systems focussing on XML view (blue part in Figure 2), we have implemented a general and extensible tool which supports almost any kind of data format.

3 THE DAEMONX FRAMEWORK

The described architecture was implemented in the *DaemonX* framework. The *DaemonX* project is a plug-in-able tool for data and/or process modeling. It was developed by the *DaemonX* team at the Faculty of Mathematics and Physics of the Charles University in Prague as a student software project. The application was designed to support user-defined plug-ins which define specific functionality needed by the author of the plug-in. These plug-ins are then managed by the application core to provide inter-operability and evolution process between models.

The key abilities of *DaemonX* are as follows:

- Support for user-defined plug-ins – for model and evolution process
- Support for core functionalities for plug-in inter-operability
- Evolution process management of the models defined in user-defined plug-ins
- Multiple views (diagrams) of the model
- Multi-diagram undo/redo management

3.1 Architecture

The core of *DaemonX* is based on the *Meta-Object Facility* (MOF) approach [16], a standard for model-driven engineering. This architecture pattern provides the ability to define a model at different layers of abstraction. *DaemonX* itself defines the meta-model (*M2 layer* in MOF). This M2 layer model is used by *DaemonX* core as an abstraction over all model plug-ins which the application controls. In the M2 layer there are defined basic classes *M2 Construct* and other needed structures of this layer, i.e. *M2 Property*, which represents a property of the M2 Construct and *M2 Relation*, which is a special structure representing connection between two M2 Constructs (for the full list see the developer documentations [17]). The authors of the particular model plug-in have to inherit from these structures in their model design and give special behaviors to their model. These particular models form the next layer, called *M1 layer* and represents a specific model like the UML class diagram. Finally, an instance of the particular UML class in the diagram is called *M0 layer*. The described idea with framework constructs and an example of design of the UML class diagram model plugin is depicted in Figure 3.

[3] `http://xcase.codeplex.com/`
[4] `http://exolutio.com/`

Next, the design of the framework plugins uses the *Model-View-Controller* pattern (MVC) [18]. This approach enables the design of a model (for example UML [3]) in logically separated parts. The model consists of application data; view (or multiple views) represents the output of the data for the user; controller mediates input and actions and manages the model and the view.

The framework itself is designed to be loosely coupled. Hence it is possible to substitute components of the framework with other components providing different functionality. The basic framework architecture diagram is depicted in Figure 4.

The framework consists of the three base parts – Core, Evolution Manager (described in Section 3.3) and Undo-Redo Manager (see Section 3.4).

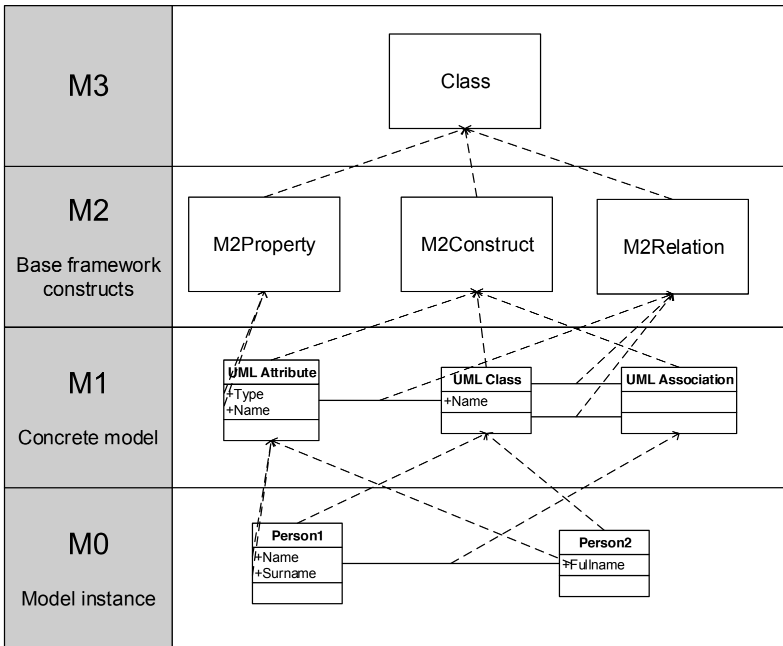

Figure 3. A schema of MOF layers and a UML example

3.2 Plug-in Support

As we have mentioned, the strength of *DaemonX* is based on the idea of plug-ins. Basically, the application provides two types plug-ins: 1) *modeling plug-in* and 2) *evolution plug-in*.

Modeling Plug-in. This type of plug-in defines a specific model which will be used by designer (for example the UML class diagram model). In the plug-in there are defined all behaviors and operations of the model. There are no limitations

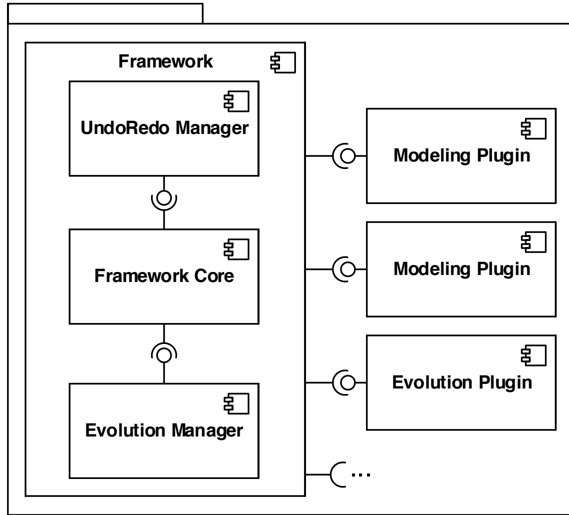

Figure 4. A simple architecture diagram

of the plug-in abilities except for *DaemonX* interfaces which the plug-in has to implement.

Evolution Plug-in. The main purpose of this plug-in is to support one-directional evolution process between two particular modeling plug-ins. As in the modeling plug-in, there are no limitations of the functionality except for implementation of the required interfaces. The next restriction is that the plug-in is related to two modeling plug-ins, called *source* and *target*.

3.3 Evolution Process

The most important and novel part of *DaemonX* is the evolution process which is ensured by the framework part called *evolution manager*. Evolution manager controls the evolution plug-ins defining one-directional description of propagation changes from a particular source model to a particular target model. This means that the plug-in knows about all public operations of both the modeling plug-ins.

Next important thing defined by the evolution plug-in are so-called *evolution references* which define how constructs from one model can be related with constructs from another model (e.g., that a UML class can be related only to a UML class, but not to a UML class attribute). Hence, there can be defined a specific behavior of the evolution process between the models. The evolution process is based on an analysis of the operations done in source modeling plug-ins. The result of the operation analysis is a collection of operations generated by the evolution plug-in which must be processed in the target models. These changes can be subsequently

set as an input of another evolution plug-in to be propagated transitively to other related model(s). A simple process diagram is depicted in Figure 5.

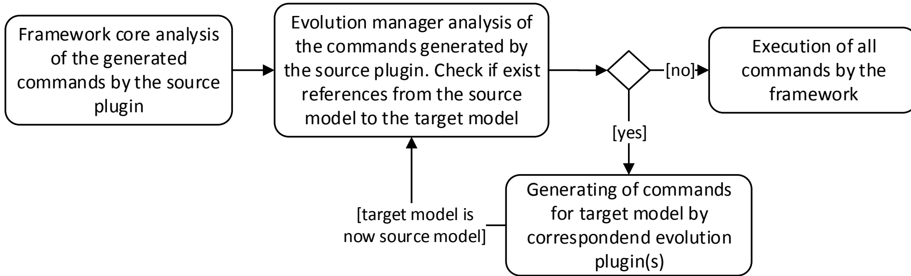

Figure 5. Evolution process diagram

The framework core and the evolution manager are loosely coupled, so its implementation can be easily modified. The current implementation of the evolution manager supports transitive propagation in a tree graph which satisfies that one diagram can be changed by the evolution process only once and the process can not get into an infinite loop. The evolution process is semi-automatic. Propagation is done automatically and the user is asked only if the validations processed before propagation (while generating operations for target model) fail or if the plugin needs some user interaction.

The first release of *DaemonX* contained the basic core framework with experimental implementation of the following model plug-ins:

- *PIM Model* for modeling of the problem domain,
- *XSEM PSM Model* for modeling of XML data,
- *UML Class Model* for modeling general data structures,
- *Relational model* for modeling relational data, and
- *BPMN Model* for modeling business processes.

And the following evolution plug-ins between existing model plug-ins were involved:

- *UML* $\rightarrow$ *UML*,
- *PIM* $\rightarrow$ *XSEM PSM*,
- *XSEM PSM* $\rightarrow$ *PIM*, and
- *PIM* $\rightarrow$ *relational model.*

In Figure 6 there are depicted screen shots of the application and models of the implemented modeling plug-ins. On the left we can see the PIM model, in the top-center is the DB PSM (relational) model, in the bottom-center the UML class model and finally on the right the XSEM PSM model.

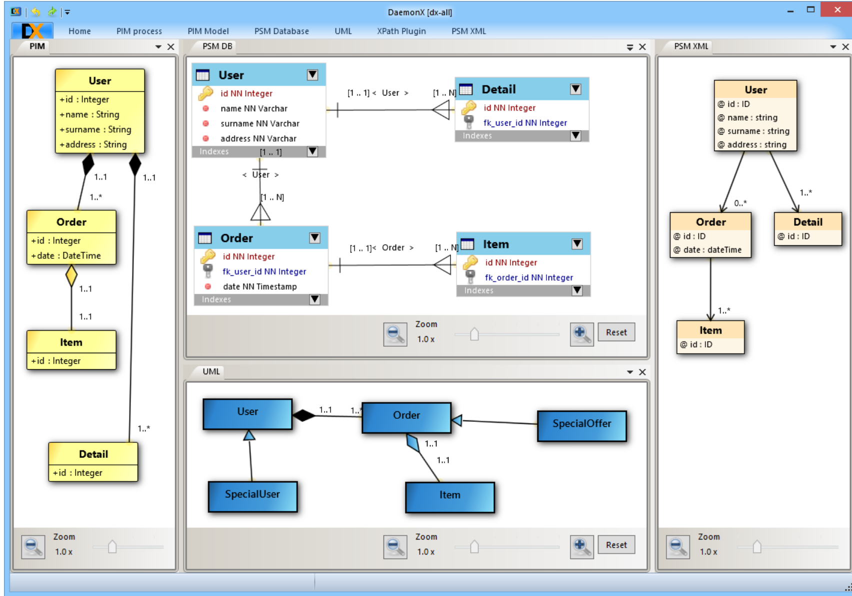

Figure 6. An example of existing modeling plugins

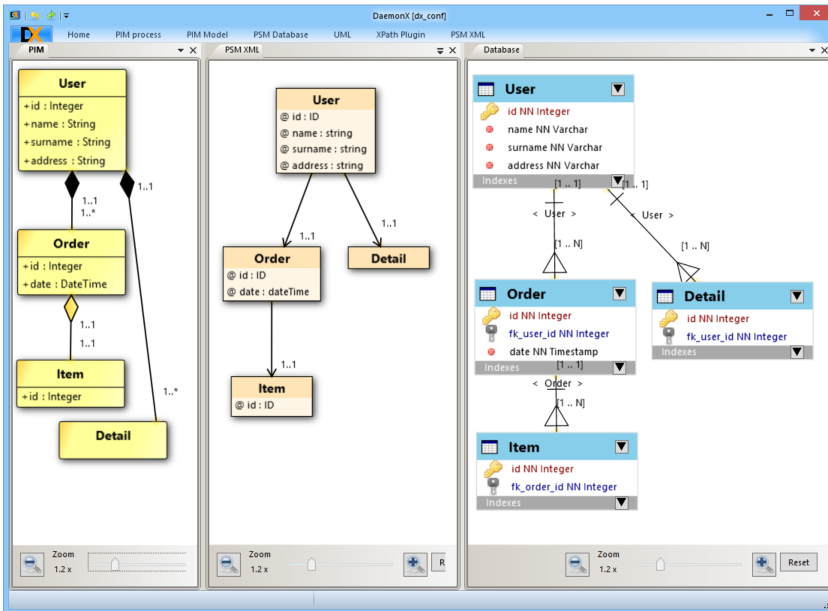

Figure 7. An example of evolution process – initial state

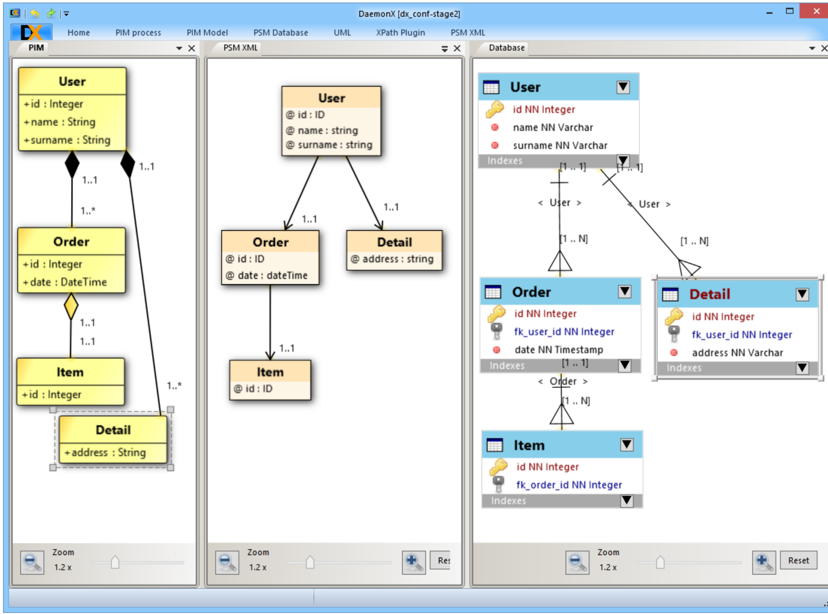

Figure 8. An example of evolution process – attribute movement

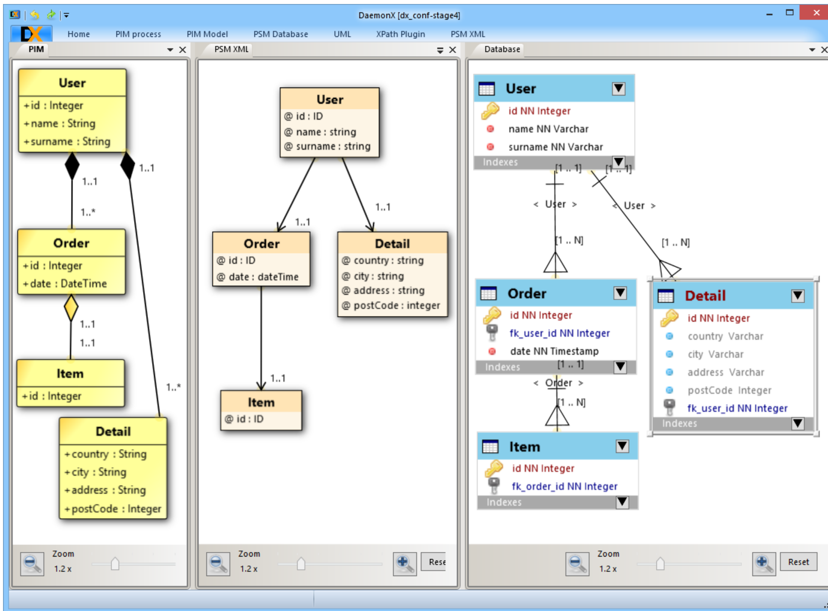

Figure 9. An example of evolution process – final state

In Figure 7 we can see an example of three models of an e-shop model design. On the left there is a PIM model representing base entities of the order, in the middle there is an XSEM PSM model representing message format and on the right there is a DB PSM model representing database schema. All these models are interconnected by references – both XSEM PSM and DB PSM are related on PIM model – they are targets of the source PIM model. Now suppose that there is a design request to move attribute *address* from entity *User* to entity *Detail* and split this attribute to new attributes *country*, *city*, *address* and *postCode* with appropriate data types. The benefit of relations between models is that the designer has to update only the source PIM model and all related models will be updated automatically.

In the first step attribute *address* is moved to entity *Detail* as we can see in Figure 8. The related attributes in the XSEM PSM and column in the DB PSM are moved automatically by the evolution process. In the second step the ability of the PIM model to split attribute *address* to multiple attributes is used. The last step is to change the data type of the new created attribute *postCode* from default type *String* to *Integer*. The final form of the models is depicted in Figure 9.

As mentioned before, the propagation is possible thanks to relations between models, especially between constructs of the model (e.g., a UML class or a UML association). In *DaemonX* there exists a special view where it is possible to manage appropriate references between constructs; the respective screen shot is shown in Figure 10. It is divided into three parts. On the left we can choose the source diagram, on the right the target diagram and in the middle an available evolution plugin between these two diagrams. Lines between constructs of these models represent created evolution references.

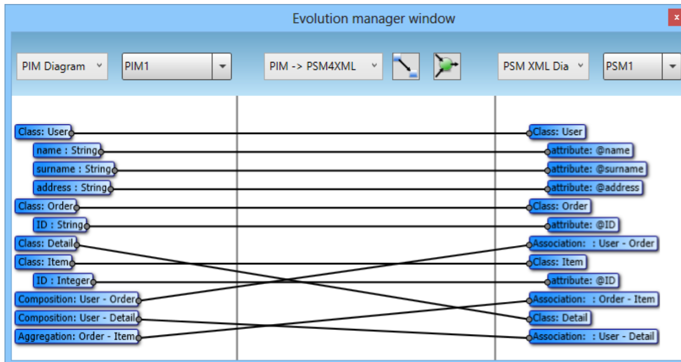

Figure 10. Screen shot of *DaemonX* – evolution manager window

3.4 Undo-Redo Management

Another ability of *DaemonX* is that it is fully command-based. All operations which are done by the user with the model or view must be defined as commands

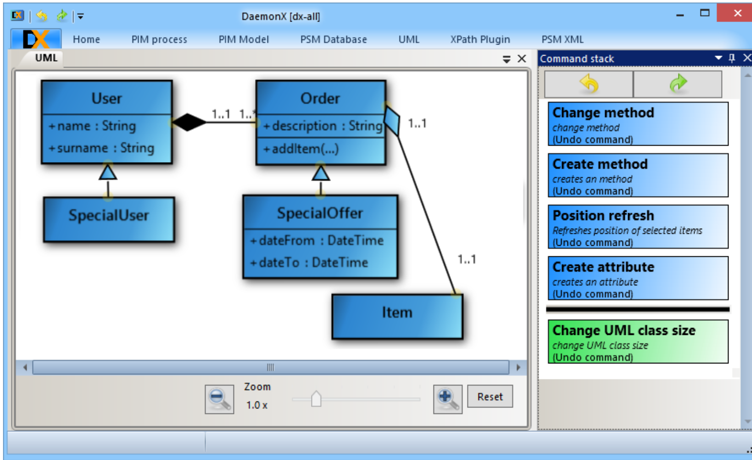

Figure 11. Screen shot of *DaemonX* with command stack for undo-redo management

in the model plug-in. This ensures the ability to provide full undo/redo support by framework part called *undo-redo manager*. As evolution manager, it is also loosely coupled with *DaemonX*.

In Figure 11 we can see a screen shot of *DaemonX*. On the right-hand side a *command stack* for undo-redo management is situated. Commands which can be undone are marked with blue color and commands which can be redone are marked with green color.

3.5 Additional Framework Extension

Since the first release, *DaemonX* was significantly extended within multiple Master Theses which enrich the first release application core and/or new add modeling and evolution plug-ins. Thesis [19] added the support for constraint languages over various models. Thesis [20] deals with propagation of changes from XML Schema model to XPath model, i.e., from XML schemas to XML queries (described in Section 4). Thesis [21] focuses on propagation of changes from relational model to SQL model, i.e., from relations to SQL queries (described in Section 5). Thesis [22] extends the undo/redo capabilities with three new algorithms. And, last but not least, thesis [23] adds the support for propagation changes among business process models. We will describe selected extensions, namely those related to evolution management of operations, in more detail in the following sections.

4 EVOLUTION OF XML SCHEMAS AND QUERIES

Thesis [20] (published in [24]) studies the related problem of *evolution* and *adaptability* of XML-based applications. One particular issue that has received much

attention is the evolution of XML queries. In the context of Figure 2, we focus on change propagation from the schema level to the operational level of the XML view (blue part) – see Figure 12.

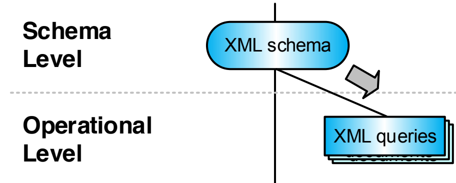

Figure 12. *DaemonX* – evolution of XML schemas and XML queries

The work was motivated by the observation that a schema change can cause not only *data inconsistency*, but also *query inconsistency*, i.e., the XML queries over the original schema and over the new schema may return different results.

To ensure correct change propagation we first need to define a model of XML schema and XPath queries and respective edit operations. Then, we can study the impact of the schema changes on the queries. In this section we show cases when the query can be directly adapted, when user interaction is required and when the adaptation cannot be decided. For our purposes we have selected only a subset of XML Schema and XPath.

4.1 Models for XML Schema and XPath

The *platform-specific model (PSM)* of the platform-specific level enables us to specify how a part of the reality modeled at the platform-independent level is represented in a particular XML schema. In addition, the designer works in a UML-style way which is more user-friendly than editing the XML schema. The model we use is called *XSEM* [13].

Definition 1. A *PSM schema* is a tuple $\mathcal{S} = (\mathcal{S}_c, \mathcal{S}_a, \mathcal{S}_r, \mathcal{S}_e, C_{\mathcal{S}}, \mathit{content})$. $\mathcal{S}_c$, $\mathcal{S}_a$, and $\mathcal{S}_r$ are sets of *classes*, *attributes*, and *associations*, respectively. $\mathcal{S}_e$ is a set of *association ends*. An association is an ordered pair $R = (E_1, E_2)$, where E_1, $E_2 \in \mathcal{S}_e$ are different association ends. Any two associations are disjoint. $C_{\mathcal{S}} \in \mathcal{S}_c$ is a *schema class* of $\mathcal{S}$. Function *content* assigns a class C with an ordered sequence of all associations with C as the parent.

A PSM schema is displayed as a UML class diagram in an ordered tree layout which reflects the hierarchical structure of XML data. Note that we omit names, types and cardinalities from the definition for simplicity. We do not cover all the schema constructs – they are covered in the full definition of our model [13]. An actual XML schema can be automatically generated from our PSM schema and vice versa. A sample self-explanatory PSM schema is depicted in Figure 13.

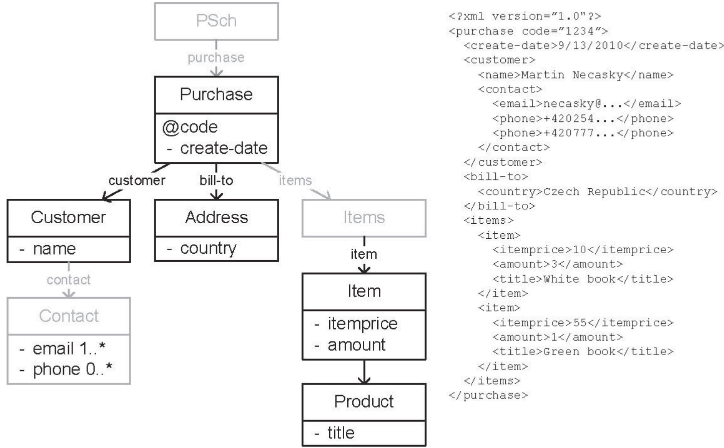

Figure 13. Sample PSM schema

For the purpose of evolution of XPath queries related to XML schemas, there must exist a mapping between an XML schema and an XPath query. Since the full XPath syntax is too extensive, we use its subset based on the *Positive Core XPath* [25] with several modifications. Our syntax at the current stage does not consider predicates and we add the operator *except* to the definition.

$$
\begin{aligned}
\mathbb{X} &\equiv \mathbb{X}|\mathbb{X} \parallel /\mathbb{X} \parallel \mathbb{X}/\mathbb{X} \parallel (\mathbb{X}) \parallel \mathbb{X} \text{ except } \mathbb{X} \parallel \mathbb{A} :: \mathbb{L} \\
\mathbb{A} &\equiv \textit{self} \parallel \textit{child} \parallel \textit{parent} \parallel \textit{descendant} \parallel \textit{ancestor} \parallel \textit{preceding} \parallel \textit{following} \\
& \quad \parallel \textit{descendant-or-self} \parallel \textit{ancestor-or-self} \parallel \textit{preceding-sibling} \parallel \textit{following-sibling}
\end{aligned}
$$

where $\mathbb{X}$ denotes a *location path* and $\mathbb{A}$ represents an *axis*. As we can see, the only one node test is possible – *name test*, denoted $\mathbb{L}$.

The original Positive Core XPath definition contains predicates, but it can only be used to test element/attribute occurrence. A query using predicates can be rewritten to a query without them and still returns the same *result set* [26]. This solution has only one problem – the query is transformed to a complex form, not transparent for the designer at first sight. In the defined syntax, it is also possible to use all classical XPath abbreviations for axes, such as "*" for all named child nodes, " " for the `child::` axis, "." for the `self::node()` step, ".." for the `parent::node()` step and "//" for `/descendant-or-self::node()/` query.

To be able to map an XSEM PSM diagram to an XPath query, an *XPath model* must be defined. We propose a model that follows the ordered tree structure of the XPath query, it results from the presented syntax and it visualizes its textual representation. The components of the model can be divided into two parts – *nodes* which represent nodes in the location path and *edges* that represent axes. An edge

and a node together comprise a *location step* of the XPath query. The model contains the following components:

- *Node* (E) – representing node test, or name test if name is specified
- *Axes* child (L_{ch}), descendant (L_d), descendant-or-self (L_{dos}), parent (L_{pa}), ancestor (L_a), ancestor-or-self (L_{aos}), following (L_f), following-sibling (L_{fs}), preceding (L_{pr}), preceding-sibling (L_{prs}) and self (L_s)
- *Expression node* (E_{ex}) representing *disjunction* (denoted by '|') and *except* operators. Its first output edge represents the first part of the expression, the second output edge represents the second part of the expression. The third edge represents the following part of the query in the sense of:
  (*first_expression operator second_expression*)/*third_expression*

An expression node with the corresponding query is visualized in Figure 14. In particular, we use the notation we have proposed in [20] and implemented in [17].

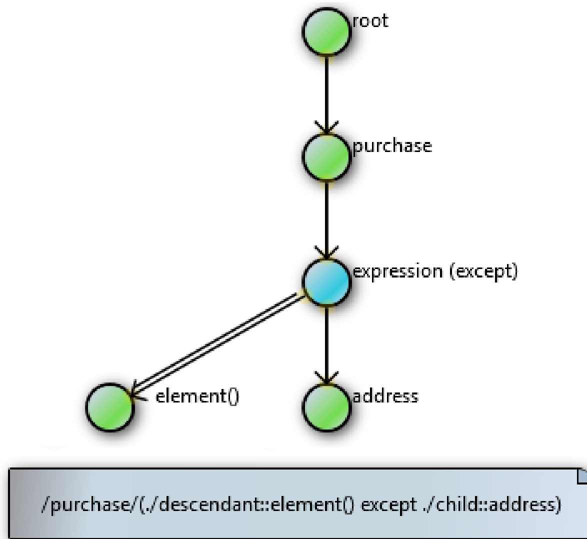

Figure 14. XPath expression node

Since the XSEM PSM schema has a tree structure and the XPath query follows a tree structure, it is straightforward and natural to map XSEM PSM to a location path. An example is shown in Figure 15; its formal definition is provided in [20]. As we can see, an axis can intervene not only a single node in the schema tree, but also a part of a tree – axis *element()* in the XPath query intervene two nodes in XSEM PSM model, *Item* and *Product*. We say that the part of the tree is *hit* by the location step. When the schema evolves, the query is gradually evaluated and

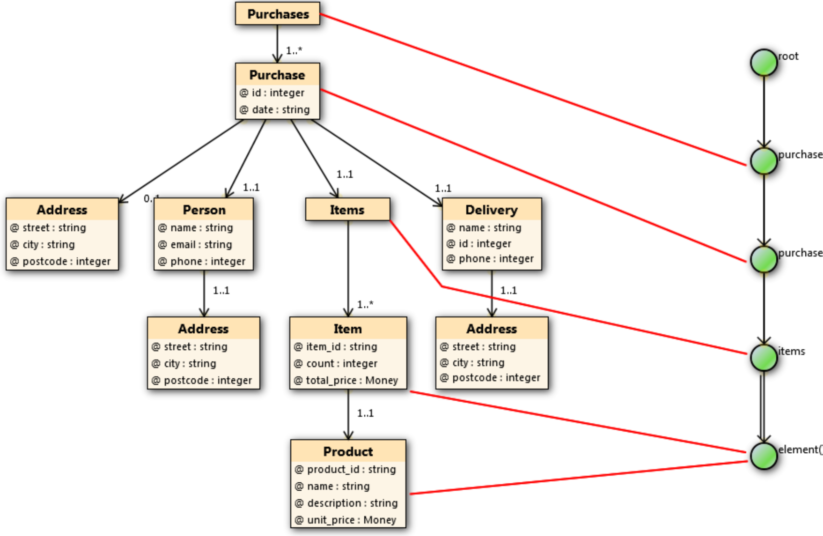

Figure 15. Mapping between XSEM and XPath models

the hit parts are compared with the previous version. If a difference is discovered, the evolution algorithm is launched.

4.2 Evolution Algorithm

Every change in the source XSEM PSM can cause changes in multiple location steps of the XPath model. All operations which change the source XSEM PSM schema are *atomic*. We use the subset of operations for query adaptation identified in [9] (see Table 1). Naturally, from the atomic operations any *composite* and more user-friendly operations can be created.

Following the set of operations in Table 1, we established similar set of operations for the XPath model – see Table 2.

Formally, let Q be the original query over the original schema S, Q' be the adapted query over the evolved schema S', $R = Q(S)$ be the result set of Q over S and $R' = Q'(S')$ be the result set of Q' over S'. Let AO_{XSEM} be an atomic operation done in an XSEM PSM schema (from Table 1) which should be propagated and let OS_{XPath} be a sequence of atomic operations in an XPath model (from Table 2) which was generated from AO_{XSEM} to preserve the same results of the queries with the original and the new XSEM PSM schema. Then $R = R' = Q(S) = Q'(S') = OS_{XPath}(Q)(AO_{XSEM}(S))$ if there exists an appropriate propagation algorithm which generates OS_{XPath}.

Due to space limitations, we will use some simplifications. (For full description see [20].) We will consider changes with a single class corresponding to element x

Operation	Description
$\alpha(C)$	Adds a new PSM class C as the root into the XSEM PSM diagram.
$\alpha(C, C_p)$	Adds a new PSM class C into XSEM PSM diagram as a child of parent class C_p. This operation creates an association A between these classes.
$\rho(A)$	Removes a PSM association A from XSEM PSM diagram.
$\rho(C)$	Removes a PSM class C from XSEM PSM diagram, s.t. C is a leaf node of the schema tree. When a PSM class C is deleted, all associations connected to C are deleted too.
$\delta(C, name)$	Sets a new name to PSM class C.
$\psi(C, direction)$	Moves a PSM class C to the left or to the right in the sequence of its siblings.
$\mu(C, C_p)$	Reconnects a PSM class C as child of PSM class C_p.

Table 1. Operations for the XSEM model

Operation	Description
$\alpha(E_{root})$	Adds a new root node E_{root} into XPath diagram.
$\alpha(E, E_p)$	Adds a new node E into XPath diagram as a child of node E_p.
$\alpha(L, E_p, E_{ch})$	Adds a new axis edge L between two nodes, child node E_{ch} and parent node E_p.
$\rho(L)$	Removes axis edge L from the diagram.
$\rho(E)$	Removes node E and all related axis edges.
$\delta(E, name)$	Sets a new name to node E.

Table 2. Operations for the XPath model

and its adding to the existing model. Also, if not specified otherwise, all considered elements are in a *sequence* element. In all presented situations we suppose that there exists no two sibling elements of the same name in S and in S'. In the description we will use functions with self-explanatory names, such as *parent()*, *descendant()*, *absolute_path_to()*, *absolute_path_to_previous_sibling()* etc.

In the following text we will consider cases when query consistency is violated (i.e., $R \neq R'$) and Q' needs to be adapted accordingly. Since each query can be divided into separate location steps, we can consider only one location step of the query Q.

An Example of Adding. This operation adds an element x as a child of an existing element in S. In the current location step we consider context element $p \in S$. Propagation of this operation is described in Table 3.

Consider schema in Figure 16 on the left and XPath model of $Q = /vehicle/child :: */registration_number$ in Figure 17 on the left. If element *motorcycle* is added as a child of element *vehicle* (see Figure 16 on the right), sub-query $/vehicle/child :: *$

Axis	**Description**
ancestor (-or-self), parent	Since x can be added only as a child element (see Table 1), adding x as ancestor/self/parent of p to the root will be solved in another location step.
child	If $Q = p/child :: *$, then $Q' = p/child :: *except\ absolute_path_to(x)$. *Note:* If x is inserted into choice or when $minOccurs$ of x is 0, then $Q' = Q$, i.e., no change propagation is needed.
descendant (-or-self)	If $Q = p/descendant :: *$, then $Q' = p/descendant/ :: *except\ absolute_path_to(x)$. *Note:* This modification is possible only if there exists no sibling element $q \in S'$, s.t. $name(q) = name(x)$. Otherwise, we should use function *position* which in combination with different values of $minOccurs$ disallows precise selection of x. Therefore, we assume no sibling elements in S and S' with the same name.
	If $Q = p/descendant :: str$, then revalidation is needed only when $name(x) = str$.
following (-sibling)	If $Q = p/following :: *$, then $Q' = p/following :: *except\ absolute_path_to(x)$.
preceding (-sibling)	If $Q = p/preceding :: *$, then $Q' = p/preceding :: *except\ absolute_path_to(x)$.
self	Solved in another location step.

Table 3. Refinement – adding element x

will return all elements including *motorcycle*. Hence, the location step is updated from $child :: *$ to $child :: *except/vehicle/motorcycle$ and $Q' = /vehicle/(child :: *except/vehicle/motorcycle)/registration_number$. The model of Q' is shown in Figure 17 on the right.

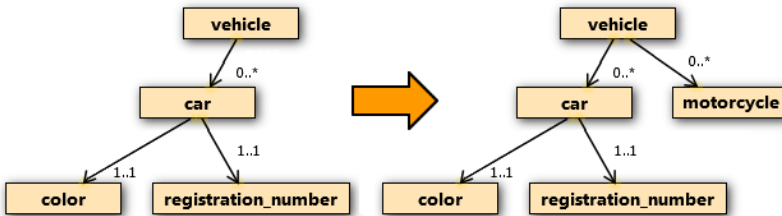

Figure 16. Schema example for adding

Description of all remaining operations on the XSEM PSM model and their propagation to XPath Query model is described in [20].

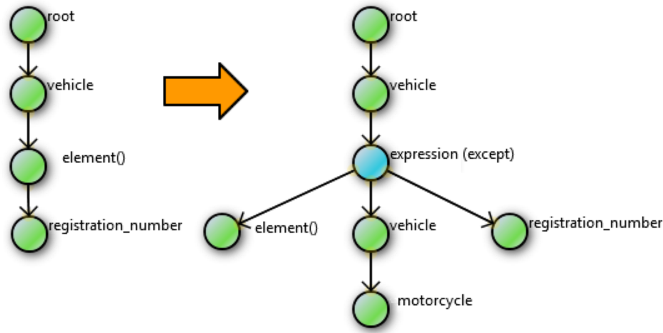

Figure 17. Query example for adding

4.3 Proof of the Concept and Implementation

The full implementation of the proposed approach was incorporated into the *DaemonX* framework. (Note that the sample figures are screen shots of the system.) Since there is no existing real-world project that provides similar abilities (see Section 6 for more details), it is not possible to compare our solution and results of others. Also a kind of quantitative evaluation is difficult, since the efficiency is not our target. Therefore, queries from *XPathMark XPath-TF* [27] were used to provide a proof of the concept, i.e., to depict that the approach works as expected. In addition, since this test set is quite simple, we also created our own more complex queries using various axes to test abilities of the solution.

Firstly, from the test set we selected tests corresponding to our XPath syntax, namely A1 – A11, P1 – P11 (rewritten into queries without predicates) and O1, O3, O4. Examples of the queries are as follows:

```
//l/ancestor::* (A5)
//l/following::* (A9)
//l/descendant::* (P5)
//l/preceding-sibling::* (P7)
//q/following::*/parent::* except //g/ancestor::* (O1)
```

All these queries were applied on the respective schema and then all possible edit operations (see Table 1) were tested. All updates were executed correctly in all cases, i.e., the data were modified according to respective mapping to queries.

Secondly, to test the approach on more complex queries, we took XML schema of an order from *Amazon AWS* [28] used for communication with customers by Web Services. XSEM PSM model was created from this schema and a set of XPath queries utilizing all available axes in various combinations was defined. These queries were automatically mapped to the schema by the *DaemonX* framework. Examples of the queries are as follows:

```
//RegionDefinition/parent::ExcludedRegions/parent::*
/Order/ParameterizedUrls/*/*
//AmazonUpsellPreferences/child::*
//ShippingRate/following-sibling::*/descendant::*
//MerchantUpsellItem/Images/preceding-sibling::*
//ShippingMethods/following::*
//RegionDefinition/ancestor::*
//Taxamount/following::Shipping/child::*
//Images/parent::*/ItemCustomDate/ancestor::Cart
```

Next, we made various changes in the schema to simulate a designer. After propagation the results of both original and new queries were checked. There were found no limitations in the evolution process.

5 EVOLUTION OF RELATIONAL SCHEMAS AND SQL QUERIES

Thesis [21] (published in [29]) also studies adaptation of queries with regard to schemas, but in different domain – it focuses on adaptation of SQL queries with regard to relational schemas. In the context of Figure 2, we have focused on change propagation from the schema level to the operational level of the storage view (green part) – see Figure 18.

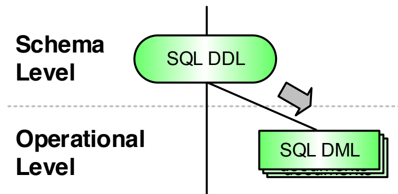

Figure 18. *DaemonX* – evolution of relational schemas and SQL queries

To describe the change propagation mechanism, we again need to define the data model, the query model, and the respective operations. The description is naturally similar to the previous case of XML data and queries; however, it corresponds to the relational model. The visualization of the models in *DaemonX* is different, since it corresponds to respective classical visualization strategies. This depicts the advantage of plug-ins and the universality of *DaemonX*. It enables to incorporate a completely different data model and respective operations and apply the same evolution management strategy.

5.1 Database Model

The database model we consider is based on the relational database model. Such database model is used as platform-specific. The PSM database model diagram

consists of *Tables*, *Columns* which are included in Tables and *Relationships* visualized as an arrow leading from the referencing table to the referenced table. The full description of the model can be found in [21]. We omit the details and examples for space limitations and general popularity of the model.

5.2 Query Model

For possibility of evolution of SQL queries related to a given database schema, there must also exist a mapping between an SQL query and a database schema model. We introduce a graph-based SQL query model which is particularly designed for the evolution process. We describe its visualization model, limitations and possibilities. The mapping between SQL query model and the database schema will be introduced as well. (The algorithm to generate the SQL query from the model can be found in [21].)

Naturally, the full SQL language syntax was not used in the proposal and for this reason there exist the following limitations:

- Projection operator '$*$' is banned to use. It is always necessary to enumerate all the columns used in the *SELECT* clause.
- In queries it is possible to use only simple column enumeration, other expressions or functions other than aggregate functions are banned to use. This limitation relates to the *CASE* construct as well.
- The model does not support *UNION*, *INTERSECT* and *EXCEPT* constructs.
- Each condition used in the SQL query is assumed to be in the *conjunctive normal form* (CNF) [30].

But, there are no major limitations to extend the work to cover wider SQL syntax and with additional propagation algorithms in future research.

The idea of the graph-based model results from papers [31] and [32]. However, it is adjusted and extended for purposes of our approach. First, each SQL query in the model is represented as a directed graph with particular properties.

Definition 2 (Query Graph.)**.** A *query graph* G of the SQL query Q is a directed graph $G_Q = (V, E)$, where V is a set of *query vertices* and E is a set of *query edges.*

Definition 3 (Query Model.)**.** A *query model* M of SQL query Q is a tuple $M_Q = (G_Q, T_V, T_E, \tau_V, \tau_E)$, where G_Q is a query graph $G_Q = (V, E)$, T_V is a set of vertex types $\{$*AggregateFunction, Alias, BooleanOperator, CombineSource, ComparingOperator, ConstantOperand, DataSource, DataSourceItem, From, FromItem, GroupBy, Having, OrderBy, OrderByType, QueryOperator, Select, SelectItem, Where*$\}$, T_E is a set of edge types $\{$*Alias, Condition, ConditionOperand, DataSource, DataSourceAlias, DataSourceItem, DataSourceParent, FromItem, FromItemParent, FromItemSource, GroupBy, GroupByColumn, Having, MapColumn, MapSource, OrderBy, OrderByColumn, SelectColumn, SelectQuery, SourceTree, Where*$\}$, function τ_V :

$V \rightarrow T_V$ assigns a type to each vertex of the query graph G_Q and function $\tau_E : E \rightarrow T_E$ assigns a type to each edge of the query graph G_Q.

A *query vertex* represents a particular part of the SQL query, e.g., a database table, a table column, a comparing operator in condition, a selected column in the *SELECT* clause, etc. A *query edge* connects parts of the SQL query together and gives a particular semantics to this connection. For instance the edge connecting a *From* vertex and a *Where* vertex means that the query contains a *WHERE* clause represented by the *Where* vertex.

Each query graph can be logically divided into smaller subgraphs. These subgraphs are called *essential components*. Each essential component has a visual equivalent in the query visualization model (described in Section 5.3). There exist the following essential components: *DataSource*, *From*, *Select*, *Condition*, *GroupBy*, *OrderBy*. For instance, the simplest SQL queries of the form *'SELECT projection FROM table'* require only *DataSource*, *From* and *Select* components. Figure 19 illustrates a simple example of the modeled *GROUP BY* clause. The example is equivalent to the following parts of the SQL query:

```
SELECT
  CustomerId as cid , COUNT(OrderId) as orderCount
  ...
GROUP BY
  CustomerId
```

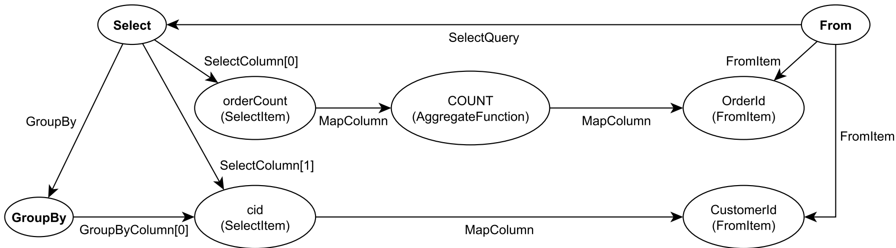

Figure 19. An example of a simple model of the *GROUP BY* clause

Note that all these components contain subcomponents. The full list and their description can be found in [21].

5.3 SQL Query Visualization Model

Although the graph-based query model can describe any SQL query, it is relatively complex. Even a query model for an extremely simple SQL query contains a lot of vertices and edges (see Figure 19). For this reason we proposed a visualization model, which simplifies a related query model for users and model creation.

The visualization model is divided into so-called *essential visual components*. Each essential component mentioned in Section 5.2 has its visual equivalent by some essential visual component, so each visual component represents a part of the SQL query graph model. We distinguish the following visual components: *DataSource*, *QueryComponent* and *Component Connection*. A *DataSource* visual component visualizes a *DataSource* essential component. For example, the rectangle *Customer* in Figure 20 shows an example of *DataSource* visual component. The example represents a database table *Customer* with columns: *customerId*, *firstname*, *lastname*, *email* and *phone*. A *QueryComponent* is a universal visual essential component, which represents parts of the SQL query: *Select*, *From*, *Where*, *GroupBy*, *Having*, *OrderBy*. For example, the rectangle *Where* in Figure 20 illustrates an example of the *QueryComponent* visual component. The example shows visualization of the *WHERE* clause. A *Component Connection* does not correspond directly to any essential component of the query graph. Instead, it covers a connection of two essential components to finish the correct and complete query graph. For example, the whole modeled SQL query in Figure 20 is the following one:

For comparison, the related query graph model consists of 45 vertices connected by 87 edges.

```
SELECT
  c.firstname, c.lastname,
  a.street, a.city, a.postcode,
FROM
  Customer as c
  JOIN Address as a
  ON c.customerId = a.customerId
WHERE
  (c.firstname = 'John'
  OR c.firstname = 'Jane')
  AND (c.lastname = 'Doe')
ORDER BY
  c.customerId ASC,
  c.lastname ASC,
  a.postcode DESC
```

5.4 Mapping to the Database Model

Since the database model consists of tables and its columns which we can interpret as a general source of data, we have a direct mapping from the database model to the query model. We do not consider database relationships between database tables in the database model. For the purpose of the query model they are not important, because the queries do not need to reflect them and join the tables arbitrarily.

The mapping between the database model and the query model is described as follows:

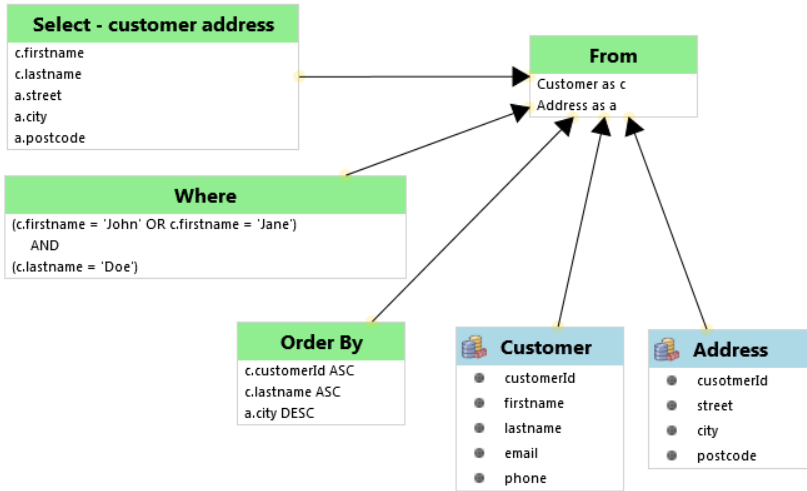

Figure 20. An example of a visual model of a more complex SQL query

- **Database table → *DataSource*** The database table in the database model is mapped to the *DataSource* visual component which corresponds to the *DataSource* vertex of the related query graph.
- **Table column → *DataSourceItem*** The table column in the database model is mapped to the *DataSourceItem* visual component which corresponds to the *DataSourceItem* vertex of the related query graph.

An example of the mapping is shown in Figure 21. The example illustrates the mapping from a database table *Customer* to a *DataSource* visual component called *Customer*.

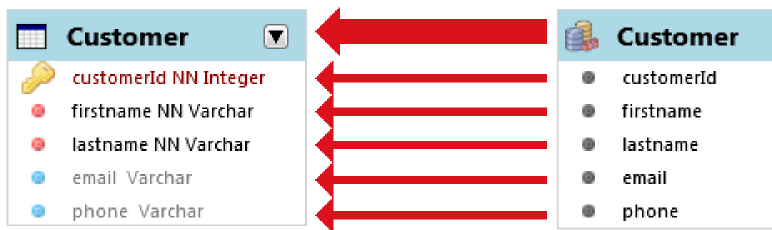

Figure 21. An example of a mapping between database model and query model

Note that the mapping does not preserve keys (primary keys, foreign keys) and any other column attributes like *NOT NULL* or *UNIQUE*. For the purpose of the data querying this property is insignificant.

5.5 Model Operations

All changes in the database model are again done via atomic operations which can form more complex operations (like, e.g., move or split). All atomic operations in the database model which have an impact on the SQL query model are translated by the evolution process into corresponding atomic operations in the SQL query model. Let us have a database model M_D, which consists of a set of tables T_i, $i \in [1, n]$. Each table $T_i \in M_D$ has a name T_{i_N} and a set T_{i_C} of columns c_j, $i \in [1, n_i]$. Each column c_j has a name c_{j_N}. The operations for modification of database model M_D are listed in Table 4.

Operation	Description
$\alpha_T : (T_i, m) \to T_i'$	Renaming Database Table: The operation returns table T_i', where $T_{i_N}' = m$ and $T_{i_C}' = T_{i_C}$.
$\beta_T : (M_D, T_i) \to M_D'$	Removing Database Table: The operation removes database table $T_i \in M_D$ from the database model M_D. It returns database model M_D', where $M_D' = M_D \backslash \{T_i\}$.
$\gamma_C : (T_i, c_j) \to T_i'$	Creating Table Column: The operation adds the column c_j into table T_i. It returns table T_i', where $T_{i_N}' = T_{i_N}$ and $T_{i_C}' = T_{i_C} \cup \{c_j\}$.
$\alpha_C : (c_j, m) \to c_j'$	Renaming Table Column: The operation returns column c_j' where $c_{j_N}' = m$.
$\beta_C : (T_i, c_j) \to T_i'$	Removing Table Column: The operation removes column $c_j \in T_i$ from the table T_i. It returns table T_i', where $T_{i_N}' = T_{i_N}$ and $T_{i_C}' = T_{i_C} \backslash \{c_j\}$.

Table 4. Operations for the database model

Let us have a query model M_Q, whose query graph G_Q consists of a set of *DataSources* D_i, $i \in [1, k]$ and other components, which are not important for our purpose. Each *DataSource* $D_i \in M_Q$ has a name D_{i_N} and a set D_{i_I}, which is a set of *DataSourceItems* d_j, $j \in [1, k_i]$. Each *DataSourceItem* d_j has a name d_{j_N}. The operations for SQL query model M_Q are listed in Table 5. Note that they are triggered by operations performed by the user over the database model M_D (as specified in Table 4). More complex operations like *Split*, *Merge*, *Move* done in the database model can be propagated to the SQL query model as well. In the SQL query model these operations are simply composed from the mentioned atomic operations.

As mentioned before, the SQL query model operations defined in Table 5 are atomic operations, i.e., they cannot be divided into smaller operations. However, in fact, each of these atomic operations over the SQL query model M_Q corresponds to a set of smaller steps called *graph operations*, which modify the query graph of the query model M_Q. In the following definitions G_Q represents a query graph $G_Q = (V, E)$. The respective operations are listed in Table 6.

Operation	**Description**
$\alpha_D : (D_i, m) \rightarrow D'_i$	Renaming *DataSource*: The operation returns *DataSource* D'_i where $D'_{i_N} = m$ and $D'_{i_I} = D_{i_I}$.
$\beta_D : (M_Q, D_i) \rightarrow M'_Q$	Removing *DataSource*: The operation removes *DataSource* $D_i \in M_Q$ from the query model M_Q. It returns query model M'_Q where $M'_Q = M_Q \backslash \{D_i\}$.
$\gamma_I : (D_i, d_j) \rightarrow D'_i$	Creating *DataSourceItem*: The operation adds *DataSourceItem* d_j into *DataSource* D_i. It returns the *DataSource* D'_i where $D'_{i_N} = D_{i_N}$ and $D'_{i_I} = D_{i_I} \cup \{d_j\}$.
$\alpha_I : (d_j, m) \rightarrow d'_j$	Renaming *DataSourceItem*: The operation returns *DataSourceItem* d'_j where $d'_{j_N} = m$.
$\beta_I : (D_i, d_j) \rightarrow D'_i$	Removing *DataSourceItem*: The operation removes *DataSourceItem* $d_j \in D_i$ from the *DataSource* D_i. It returns *DataSource* D'_i where $D'_{i_N} = D_{i_N}$ and $D'_{i_I} = D_{i_I} \backslash \{d_j\}$.

Table 5. Operations for the query model

These atomic functions are combined in the set of so-called *composite operations* which are called in the evolution process. All these functions with theirs algorithms are described in [21].

An Example of Creating a Table Column. First, Algorithm 1 creates a new *FromItem* vertex in the corresponding *From* vertex and connects it with appropri-

Operation	**Description**
$\gamma_v : (G_Q, v) \rightarrow G'_Q$	CreateVertex: The operation returns graph $G'_Q = (V \cup \{v\}, E)$.
$\gamma_e : (G_Q,\, v_{src},\, v_{dst},\, e_{type}) \rightarrow G'_Q$	CreateEdge: The operation creates edge $e = (v_{src}, v_{dst})$ such that $EdgeType(e) = e_{type}$ and returns graph $G'_Q = (V, E \cup \{e\})$.
$\beta_v : (G_Q, v) \rightarrow G'_Q$	RemoveVertex: The operation returns graph $G'_Q = (V \backslash \{v\}, E \bigcap \binom{V \backslash \{v\}}{2})$.
$\beta_e : (G_Q, e) \rightarrow G'_Q$	RemoveEdge: The operation returns graph $G'_Q = (V, E \backslash \{e\})$.
$\lambda : (G_Q, v, l) \rightarrow G'_Q$	ChangeLabel: The operation returns graph $G'_Q = (V \backslash \{v\} \cup \{v'\}, E)$, where vertex type $v'_{type} = v_{type}$ and label $v'_L = l$.
$\eta : (G_Q, C, t) \rightarrow G'_Q$	ChangeConnectionType: The operation returns graph $G'_Q = (V \backslash \{C\} \cup \{C'\}, E)$ where *CombineSource* vertex C' corresponds to vertex C with connection type t.

Table 6. Operations for the query graph

ate vertices. Subsequently it traverses to the *Select* vertex, where it creates corresponding vertices and edges using algorithm *DistributeCreatingDatasourceItemSelect* (see [21]). Finally it traverses to the *OrderBy* vertex, where it creates corresponding vertices and edges using algorithm *DistributeCreatingDatasourceItemOrderBy*. Figure 22 depicts adding of a new *DataSourceItem OrderDate* to the *DataSource* component using Algorithm *DistributeCreatingDatasourceItem* and to the *From* component using Algorithm 1. In the figure the original elements are black and the new elements of the query graph are highlighted with a red color.

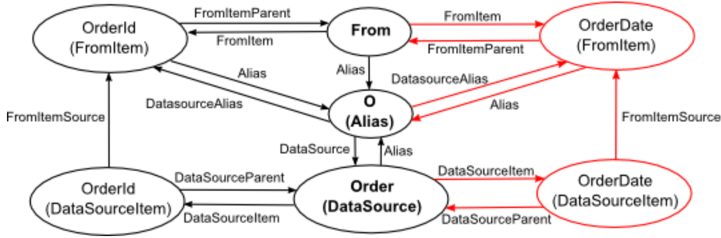

Figure 22. An example of adding new *DataSourceItem* to the *DataSource* and to the *From* components

---

Algorithm 1 DistributeCreatingDatasourceItemOnAlias

Require: *Alias* vertex A, change context C
Ensure: Graph operations to create new *DataSourceItem*.

1: $fromVertex \leftarrow A.GetNeighbour(DataSourceAlias)$
2: $newItem \leftarrow$ new $FromItem(C.Name)$
3: $C.Plan \leftarrow CreateVertex(C.G_Q, newItem)$
4: $C.Plan \leftarrow CreateEdge(C.G_Q, C.Originator, newItem, FromItemSource)$
5: $C.Plan \leftarrow CreateEdge(C.G_Q, newItem, A, Alias)$
6: $C.Plan \leftarrow CreateEdge(C.G_Q, fromVertex, newItem, FromItem)$
7: $C.Plan \leftarrow CreateEdge(C.G_Q, newItem, fromVertex, FromItemParent)$
8: $C.Originator \leftarrow newItem$
9: $selectVertex \leftarrow fromVertex.GetNeighbour(SelectQuery)$
10: **if** $selectVertex != \text{null}$ **then**
11: $\quad DistributeCreatingDatasourceItemOnSelect(selectVertex, C)$
12: **end if**
13: $orderByVertex \leftarrow fromVertex.GetNeighbour(OrderBy)$
14: **if** $orderByVertex != \text{null}$ **then**
15: $\quad DistributeCreatingDatasourceItemOnOrderBy(orderByVertex, C)$
16: **end if**
17: $C.Plan \leftarrow ResetContent(C.G_Q, fromVertex)$

Algorithm *DistributeCreatingDatasourceItemSelect* creates a new *SelectItem* vertex in the *SELECT* clause. Then it checks, whether the *GroupBy* vertex exists. If

it does, it connects the *GroupBy* vertex with the new *SelectItem* vertex. Finally it traverses to all Alias vertices of dependant queries and applies already mentioned Algorithm 1.

5.6 Proof of the Concept and Implementation

Again the full experimental implementation of the presented approach was incorporated into *DaemonX*. (In fact, Figures 20, 23 and 24 are screen shots of the application.) Our approach adds two new plug-ins into *DaemonX*. The first plug-in is a plug-in for SQL query modeling described in Section 5.3 (a screen-shot from this plug-in is depicted in Figure 20). The second plug-in is used for evolution propagation from the PSM database model to the SQL query model.

Since there are no existing applications which provide comparable features[5], to be able to depict the features of our approach we used an existing database project *Adventure Works* [34]. In the database model we modeled a set of tables of a given database schema (the list of them can be found in [21]). From this database model we derived to the query model tables as *DataSources*, which we used to model queries and views. Next, we applied various operations over the database model to simulate propagation to the query model. After the propagation the new queries were inspected whether they correspond to the expected results.

Due to space limitations we will use some simplification. We present an example of adding new column to the table and its propagation to the related *GroupBy* query. Suppose the following SQL query which model is depicted in Figure 23:

```
SELECT
   d.GroupName, COUNT(d.Name)
   as NumberOfDepartments
FROM
   HumanResources.Department as d
GROUP BY
   d.GroupName
HAVING COUNT(d.Name) > 2
ORDER BY
   NumberOfDepartments DESC
```

A new column *GroupID* was added to the table *HumanResources.Department*. This change was propagated into the complex *GroupBy* query. The new column was added to all its components. The original query was transformed by algorithm *DistributeCreatingDatasourceItem* (described in [21]) to the new query (its model is depicted in Figure 24):

[5] Though there are projects which focus on similar areas, such as [33] – a more detailed description can be found in Section 6.

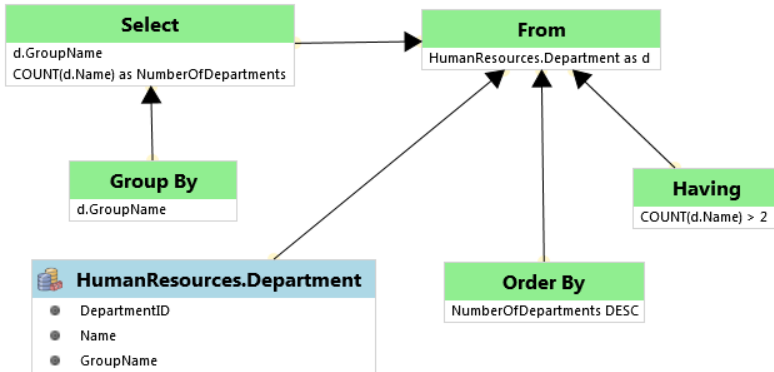

Figure 23. The model of the complex usage of complex *GroupBy* query

```
SELECT
   d.GroupName, COUNT(d.Name)
   as NumberOfDepartments,
   d.GroupID
FROM
   HumanResources.Department as d
GROUP BY
   d.GroupName, d.GroupID
HAVING COUNT(d.Name) > 2
ORDER BY
   NumberOfDepartments DESC,
   d.GroupID ASC
```

6 RELATED WORK

The current approaches towards evolution management can be classified according to distinct aspects [35]. The changes and transformations can be expressed [36] as well as divided [37] variously too. However, to our knowledge there exists no general framework comparable to our proposal made in Section 2; 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 from the point of view of Figure 2.

XML View. Current approaches of the XML view consider changes at the schema level and differ in the selected XML schema language, i.e., DTD [38, 39] or XML Schema [40, 41]. In general, the transformations can be variously classified. For

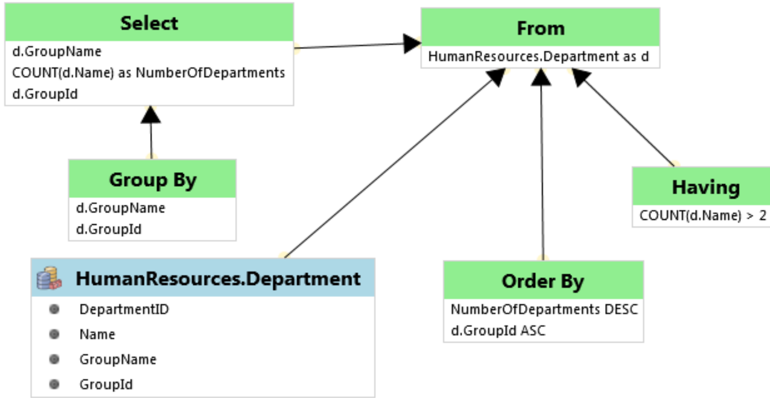

Figure 24. The updated model of the complex *GroupBy* query

instance, paper [40] 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 [41] a language called *XSUpdate* is described. The changes are then automatically propagated to the extensional level to ensure validity of XML data. There also exists an opposite approach that enables to evolve XML documents and propagate the changes to their XML schema [42]. Other approaches are similar, but they consider changes at an abstraction of logical level – either visualization [43] or a kind of UML diagram [44]. 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. Unfortunately, the amount of existing works is relatively low. The paper [45] gives recommendations on how to write queries that do not need to be adapted for an evolving schema. On the other hand, in [46] 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 [47] 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 [48].

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* [49]. However, this is not the area we are dealing with since the updates are mostly considered within the respective XML schema. In the area of evolution of general database schemas we can find approaches that focus on evolution of (object-)relational schemas [50, 51] as well as object-oriented schemas [52, 53]. 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 [54], i.e., schema level or propagation to an operational level [50].

In the purely XML-related approaches we need to consider *schema-driven* storage strategies. As surveyed in [55], 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*[6], *IBM DB2*[7] and *Microsoft SQL Server*[8]. 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 to provide such transformation script semi-automatically [56].

Processing View. Since we are considering the area of evolution of XML applications, we cannot omit the most popular application 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 [57]. In [58] the authors describe a plug-in to IBM *Rational Software Architect* (RSA)[9] which enables semi-automatic propagation of changes from business process model of Web Services 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 is used in system *Morpheus* [59], 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.

In [60] the authors solve the problem using a completely different strategy. They provide an *abstract service definition model* (ASD) which enables us to model all

[6] http://www.oracle.com/us/products/database/
[7] http://www-01.ibm.com/software/data/db2/
[8] http://www.microsoft.com/sqlserver/2008/en/us/
[9] http://www-01.ibm.com/software/awdtools/architect/swarchitect/

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 [61] the authors model the Web Services using Formal Concept Analysis and, in particular, lattices or in [62] 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.

7 CONCLUSION

In this paper we have introduced *DaemonX*, an implementation of a five-level evolution framework we have proposed in our previous papers [8, 9]. Our aim was to create a robust and extensible evolution framework that enables to concurrently model all related parts of the system (i.e., data structures, ICs and operations) as precisely as possible (i.e., with a rich set of constructs), to preserve the relations between system components, and to enable respective change management (i.e., correct propagation of changes to all the affected parts). First we have briefly described the idea of a five-level evolution framework and its advantages. The core of the text, however, considered *DaemonX*. We have described the way the framework is implemented and the way it can be extended with other functionality. We have also provided several screen shots demonstrating its interface. Then we have focused on two extensions of the framework (1) evolution of XML schemas and consequent change management of XML queries and (2) evolution of relational schemas and consequent change management of SQL queries.

Even though the tool currently covers the key aspects so that it can be used in most applications and, at the same time, proves the proposed concepts from various points of view (i.e., various data and processing models), there is a significant amount of open problems and future directions we want to focus on. The key areas involve support for more complex constructs of the implemented models, support for integrity constraints, and business process modeling.

Acknowledgement

This work was partially supported by the Czech Science Foundation (GAČR), grant number 202/11/P455 and the Grant Agency of the Charles University (GAUK), grant number 1416213.

REFERENCES

[1] IBM. Service Oriented Architecture – SOA. IBM. `http://www-01.ibm.com/software/solutions/soa/`.

[2] W3C. Web Services Activity, 2009. http://www.w3.org/2002/ws/.

[3] Object Management Group. UML Superstructure Specification 2.1.2, November 2007. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/.

[4] Chen, P. P.: The Entity-Relationship Model – Toward a Unified View of Data. ACM Transactions on Database Systems, Vol. 1, 1976, No. 1, pp. 9–36.

[5] OMG. Documents Associated with Business Process Modeling Notation (BPMN) 1.2. OMG, 2009. http://www.omg.org/spec/BPMN/1.2/.

[6] W3C. XML Path Language (XPath) Version 2.0, W3C Working Draft. 15 November 2002. http://www.w3.org/TR/xpath20/.

[7] ISO/IEC 9075-1:2008. Part 1: Framework (SQL/Framework). Int. Organization for Standardization, 2008.

[8] Nečaský, M.—Klímek, J.—Malý, J.—Mlýnková, I.: Evolution and Change Management of XML-Based Systems. Journal of Systems and Software, Vol. 85, 2012, No. 3, pp. 683–707.

[9] Nečaský, M.—Mlýnková, I.—Klímek, J.—Malý, J.: When Conceptual Model Meets Grammar: A Dual Approach to XML Data Modeling. International Journal on Data & Knowledge Engineering, Vol. 72, 2012, pp. 1–30.

[10] Boag, S.—Chamberlin, D.—Fernández, M. F.—Florescu, D.—Robie, J.—Siméon, J.: XQuery 1.0: An XML Query Language. W3C, 2007. http://www.w3.org/TR/xquery/.

[11] ISO. ISO/IEC 9075-14:2003 Part 14: XML-Related Specifications (SQL/XML). ISO, 2006.

[12] Miller, J.—Mukerji, J.: MDA Guide Version 1.0.1. Object Management Group, 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

[13] Nečaský, M.: Conceptual Modeling for XML, Volume 99 of Dissertations in Database and Information Systems. IOS Press, Amsterdam, Netherlands, 2009.

[14] Chen, P. P.: The Entity-Relationship Model: Toward a Unified View of Data. ACM Transactions on Database Systems, Vol. 1, 1976, No. 1, pp. 9–36.

[15] Thompson, H. S.—Beech, D.—Maloney, M.—Mendelsohn, N.: XML Schema Part 1: Structures (Second Edition). W3C, October 2004. http://www.w3.org/TR/xmlschema-1/.

[16] OMG. Meta-Object Facility. http://www.omg.org/mof/.

[17] DaemonX-Team. Daemonx, June 2001. http://daemonx.codeplex.com.

[18] Weisfeld, M.: The Object-Oriented Thought Process. Developer's Library. Addison-Wesley, 2009.

[19] Piják, P.: Universal Constraint Language. Master's thesis, Charles University in Prague, 2011. http://www.ksi.mff.cuni.cz/~holubova/dp/Pijak.pdf.

[20] Polák, M.: XML Query Adaptation. Master's thesis, Charles University in Prague, 2011. http://www.ksi.mff.cuni.cz/~holubova/dp/Polak.pdf.

[21] Chytil, M.: Adaptation of Relational Database Schema. Master's thesis, Charles University in Prague, 2011. http://www.ksi.mff.cuni.cz/~holubova/dp/Chytil.pdf.

[22] Jakubec, K.: Management of Undo/Redo Operations in Complex Environments. Master's thesis, Charles University in Prague, 2012. `http://www.ksi.mff.cuni.cz/~holubova/dp/Jakubec.pdf`.

[23] Kudelas, V.: Adapting Service Interfaces when Business Processes Evolve. Master's thesis, Charles University in Prague, 2012. `http://www.ksi.mff.cuni.cz/~holubova/dp/Kudelas.pdf`.

[24] Polák, M.—Mlýnková, I.—Pardede, E.: XML Query Adaptation as Schema Evolves. Proceedings of the 21st International Conference on Information Systems Development (ISD '12), Prato, Italy, 2012, Springer Science + Business Media, Inc., 2013, pp. 401–416.

[25] Hartel, P. H.: A Trace Semantics for Positive Core XPath. TIME '05, Washington, DC, USA, 2005, IEEE, pp. 103–112.

[26] Cate, B.—Marx, M.: Axiomatizing the Logical Core of XPath 2.0. Theor. Comp. Sys., Vol. 44, 2009, pp. 561–589.

[27] Franceschet, M.: XPathMark. `http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/FT.html`.

[28] Amazon. Amazon Web Services. `http://amazonpayments.s3.amazonaws.com/documents/order.xsd`.

[29] Chytil, M.—Polak, M.—Necasky, M.—Holubova, I.: Evolution of a Relational Schema and Its Impact on SQL Queries. IDC '13, Studies in Computational Intelligence, Springer International Publishing, Vol. 511, 2013, pp. 5–15.

[30] Korovin, K.: CNF and Clausal Form. In Logic in Computer Science. Springer, 2006.

[31] Papastefanatos, G.—Vassiliadis, P.—Vassiliou, Y.: Adaptive Query Formulation to Handle Database Evolution. ICAiSE '06, Springer, 2006, pp. 5–9.

[32] Papastefanatos, G.—Vassiliadis, P.—Simitsis, A.—Aggistalis, K.—Pechlivani, F.—Vassiliou, Y.: Language Extensions for the Automation of Database Schema Evolution. In: Cordeiro, J., Filipe, J. (Eds.): ICEIS '08, 2008, pp. 74–81.

[33] Papastefanatos, G.—Kyzirakos, K.—Vassiliadis, P.—Vassiliou, Y.: Hecataeus: A Framework for Representing SQL Constructs as Graphs. EMMSAD '05, 2005, pp. 13–17.

[34] Adventure Works Team. Adventure Works 2008R2, November 2010. `http://msftdbprodsamples.codeplex.com`.

[35] Mens, T.—Van Gorp, P.: A Taxonomy of Model Transformation. Electron. Notes Theor. Comput. Sci., Vol. 152, 2006, pp. 125–142.

[36] OMG. MOF QVT Final Adopted Specification. Object Modeling Group, June 2005. `http://fparreiras/papers/mof_qvt_final.pdf`.

[37] Cicchetti, A.—Di Ruscio, D.—Pierantonio, A.: Managing Dependent Changes in Coupled Evolution. ICMT '09, Springer, LNCS, Vol. 5563, 2009, pp. 35–51.

[38] Al-Jadir, L.—El-Moukaddem, F.: Once Upon a Time a DTD Evolved into Another DTD... Object-Oriented Information Systems, Springer, Berlin, Heidelberg, 2003, pp. 3–17.

[39] Coox, S. V.: Axiomatization of the Evolution of XML Database Schema. Program. Comput. Softw., Vol. 29, 2003, No. 3, pp. 140–146.

[40] Tan, M.—Goh, A.: Keeping Pace with Evolving XML-Based Specifications. EDBT '04 Workshops, Springer, Berlin, Heidelberg, 2005, pp. 280–288.

[41] Cavalieri, F.: EXup: An Engine for the Evolution of XML Schemas and Associated Documents. EDBT '10, ACM, New York, NY, USA, 2010, pp. 1–10.

[42] Bouchou, B.—Duarte, D.—Halfeld Ferrari Alves, M.—Laurent, D.—Musicante, M. A.: Schema Evolution for XML: A Consistency-Preserving Approach. Mathematical Foundations of Computer Science, Springer-Verlag, Prague, Czech Republic, 2004, pp. 876–888.

[43] Klettke, M.: Conceptual XML Schema Evolution – The CoDEX Approach for Design and Redesign. BTW '07, Aachen, Germany, March 2007, pp. 53–63.

[44] Domínguez, E.—Lloret, J.—Rubio, A. L.—Zapata, M. A.: Evolving XML Schemas and Documents Using UML Class Diagrams. DEXA '05, Springer, LNCS, Vol. 3588, 2005, pp. 343–352.

[45] Moro, M. M.—Malaika, S.—Lim, L.: Preserving XML Queries During Schema Evolution. WWW '07, ACM Press, New York, NY, USA, 2007, pp. 1341–1342.

[46] Geneves, P.—Layaida, N.—Quint, V.: Identifying Query Incompatibilities with Evolving XML Schemas. ICFP '09, ACM, New York, NY, USA, 2009, pp. 221–230.

[47] Passi, K.—Morgan, D.—Madria, S.: Maintaining Integrated XML Schema. IDEAS '09, ACM, New York, NY, USA, 2009, pp. 267–274.

[48] Bellahsene, Z.—Bonifati, A.—Rahm, E.: Schema Matching and Mapping. Data-Centric Systems and Applications. Springer Berlin Heidelberg, 2011.

[49] Chamberlin, D.—Florescu, D.—Melton, J.—Robie, J.—Siméon, J.: XQuery Update Facility 1.0. W3C, 2007.

[50] Curino, C.—Moon, H. J.—Zaniolo, C.: Automating Database Schema Evolution in Information System Upgrades. HotSWUp '09, ACM, New York, NY, USA, 2009, pp. 1–5.

[51] Curino, C. A.—Moon, H. J.—Zaniolo, C.: Graceful Database Schema Evolution: The PRISM Workbench. Proc. VLDB Endow., Vol. 1, 2008, No. 1, pp. 761–772.

[52] Banerjee, J.—Kim, W.—Kim, H.-J.—Korth, H. F.: Semantics and Implementation of Schema Evolution in Object-Oriented Databases. SIGMOD Rec., Vol. 16, 1987, No. 3, pp. 311–322.

[53] Lerner, B. S.: A Model for Compound Type Changes Encountered in Schema Evolution. ACM Trans. Database Syst., Vol. 25, 2000, No. 1, pp. 83–127.

[54] An, Y.—Hu, X.—Song, I.-Y.: Round-Trip Engineering for Maintaining Conceptual-Relational Mappings. CAiSE '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 296–311.

[55] Simanovsky, A. A.: Data Schema Evolution Support in XML-Relational Database Systems. Program. Comput. Softw., Vol. 34, 2008, No. 1, pp. 16–26.

[56] Malý, J.—Mlýnková, I.—Nečaský, M.: XML Data Transformations as Schema Evolves. ADBIS '11, Springer-Verlag, Vienna, Austria, 2011.

[57] Sun, C.—Rossing, R.—Sinnema, M.—Bulanov, P.—Aiello, M.: Modeling and Managing the Variability of Web Service-Based Systems. J. of Systems and Software, Vol. 83, 2010, No. 3, pp. 502–516.

[58] Sindhgatta, R.—Sengupta, B.: An Extensible Framework for Tracing Model Evolution in SOA Solution Design. OOPSLA '09, ACM, New York, NY, USA, 2009, pp. 647–658.

[59] Ravichandar, R.—Narendra, N. C.—Ponnalagu, K.—Gangopadhyay, D.: Morpheus: Semantics-Based Incremental Change Propagation in SOA-Based Solutions. IEEE International Conference on Services Computing, 2008, Vol. 1, pp. 193–201.

[60] Andrikopoulos, V.—Benbernou, S.—Papazoglou, M. P.: Managing the Evolution of Service Specifications. CAiSE '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 359–374.

[61] Aversano, L.—Bruno, M.—Di Penta, M.—Falanga, A.—Scognamiglio, R.: Visualizing the Evolution of Web Services Using Formal Concept Analysis. IWPSE '05, 2005, pp. 57–60.

[62] Stevens, P.: Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. Software and System Modeling, Vol. 9, 2010, No. 1, pp. 7–20.

Marek Polák received his M.Sc. degree in computer science from Charles University in Prague, Czech Republic in 2011. Currently he is a Ph.D. candidate at Department of Software Engineering of the Charles University in Prague. His research is focused on MDA and model evolution management. He also works as architect and developer in Puls.io project in the office in Prague, Czech Republic. Previously he was working as a system specialist in Commerzbank AG on investment banking systems for world-wide branches. He is interested in cloud services and NoSQL database.

Martin Chytil received his M.Sc. degree in computer science from Charles University in Prague, Czech Republic in 2012. Currently he works as a software developer in Infor in the office in Prague, Czech Republic. He is working on a database backend of Business Intelligence software products of Infor BI Application stack. He also participates in a development of core services for cloud architecture. His professional interests are relational databases and OLAP, especially in connection with SOA and cloud computing.

Karel Jakubec received his M.Sc. degree in computer science from the Charles University in Prague, Czech Republic in 2012. Currently he works as a system architect in Puls.io, building the best platform for API monitoring. Before he worked as a senior software engineer for GoodData, hacking various parts of the Open Analytics platform. He is interested in relational databases (mostly Postgresql) and recently discovered ease of web development with Python and Flask.

Vladimír Kudelas received his M.Sc. degree in software systems from Charles University in Prague, Czech Republic in 2012. Currently he is a team leader of hospitality project SilverPro at the NCR Corporation, Prague Center of Excellence. He is actively participating in analysis and feature specifications for this project. He was also a member of Autopilot team at the NCR Corporation working on automated testing tool for Aloha Point of Sale.

Peter Piják received his M.Sc. degree in computer science from Charles University in Prague, Czech Republic in 2011. Currently he works as a software developer in Descartes Systems Group in the office in Žilina, Slovakia. He is working on software product solving vehicle routing problem, processing map data and computing shortest paths for vehicles. He has published a publication in the area of modeling data and integrity constraints.

Martin Nečaský received his Ph.D. degree in computer science from Charles University in Prague, Czech Republic in 2008, where he currently works as Assistant Professor at the Department of Software Engineering. His research areas include linked data, semantic web, XML data design, integration and evolution. He is an organizer/PC chair/member of more than 10 international events. He has published more than 50 papers (5 of them received the Best Paper Award). He published 3 book chapters and one book. He is a founding member of the OpenData.cz initiative.

Irena Holubová received her Ph.D. degree in computer science from Charles University in Prague, Czech Republic in 2007. Currently she is Associate Professor at the Department of Software Engineering, Charles University and external member of the Department of Computer Science and Engineering, Czech Technical University. She has published more than 80 publications in the area of XML data management and web engineering; she gained 4 times the Best Paper Award. She is a PC member or reviewer of 15 international events and co-organizer of 4 international workshops.