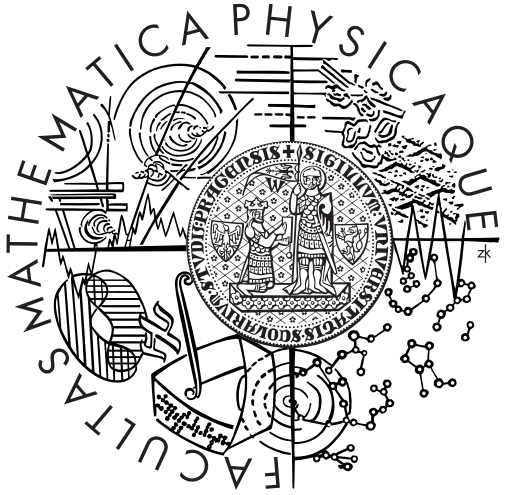
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Jakub Malý

Evoluce XML schémat

Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Irena Mlýnková, Ph.D.
Studijní program: Informatika, softwarové systémy

2010

I would like to thank to my supervisor RNDr. Irena Mlýnková, Ph.D. for her helpful suggestions, thorough notes and provided related research material. I would also like to thank to Mgr. Martin Nečaský, Ph.D. for his suggestions and comments.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

In Prague on April 12, 2010 Jakub Malý

Contents

Název práce: *Evoluce XML schémat*
Autor: *Jakub Malý*
Katedra (ústav): *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *RNDr. Irena Mlýnková, Ph.D.*
e-mail vedoucího: *mlynkova@ksi.mff.cuni.cz*

Abstrakt: *V předložené práci studujeme evoluci XML dat, a především důvody a dopady evoluce XML schémat. Práce obsahuje přehled existujících přístupů. Přístup prezentovaný v této práci rozšiřuje konceptuální model XSem o podporu více verzí systému. Díky tomuto rozšíření lze definovat sadu změn mezi dvěma verzemi. Dále práce obsahuje popis algoritmu, který porovnáním dvou verzí schématu vytvoří revalidační skript v jazyce XSL.*

Klíčová slova: *XML, modelování XML dat, evoluce XML dat, XSem*

Title: *Evoluce XML schémat*
Author: *Jakub Malý*
Department: *Katedra softwarového inženýrství*
Supervisor: *RNDr. Irena Mlýnková, Ph.D.*
Supervisor's e-mail address: *mlynkova@ksi.mff.cuni.cz*

Abstract: *In the presented work we study the XML data evolution, reasons and consequences of XML schema evolution in particular. The thesis contains a survey of the existing approaches to this problem. The approach presented in this work extends the XSem conceptual model with the support for multiple versions of the model. Thanks to this extension, it is possible to define a set of changes between two versions of a schema. The thesis contains a description of an algorithm that compares two versions of a schema and produces a revalidation script in XSL.*

Keywords: *XML, modeling XML data, XML data evolution, XSem*

Chapter 1

Introduction

1.1 Motivation

With the number of computer and Internet users growing every day the amount of data created, sent, interchanged and stored grows in an adequate rate. Beside storing the data in relational database management systems, the eXtensible Markup Language (XML) [30] is getting notable attention among the developers.

Relational databases still rule in the field of efficiency, but the records and tables are less suitable for message and data exchange. On the other hand, the XML format is easy to understand both to human users as well as parsers and applications regardless the platform and environment and can be easily transmitted over any protocol.

These are some of the reasons that lead many software designers to choose a relational database as a data storage and XML as the message format. The format of the messages is usually specified in the form of an *XML schema* using one of the languages mentioned in Section 2. Documents compliant with a certain schema are *valid*.

The XML applications are usually dynamic. During a system's life cycle and growing group of users, requests for changes in the system emerge, usually for one of the following reasons:

- changes (additions) at the conceptual level are required when a system is expanding
- existing interface needs to be extended or modified (for example to comply with a third party interface or standard)

- there are flaws in the current design that need to be corrected

These changes usually enforce changes in the structure and content of the messages used in the system and will be reflected in the changes of the XML schemas. This process is called *XML schema evolution*.

Transition to a new set of schemas can be complicated.

- If the old interface is no longer supported
  - other parties communicating with the evolved systems have to modify their components to accept messages/documents valid against evolved schemas,
  - existing documents must be modified to become valid against the new schema (and these documents may be distributed among the system users, not easily accessible for the evolved system).
- If the system maintains support for the old interface (which is usually a necessity at least for some period of time),
  - the system must be able to handle both versions of documents and messages which impairs maintainability of the system and increases its complexity.

It can be seen that system evolution can be a costly process but some of the difficulties can be significantly reduced when using a convenient tool.

1.2 XML Schema

When two participants decide to share data and communicate via XML messages, they have to agree on a certain contract defining the syntax and semantics of the messages.

The types of requirements on the *structure* of XML documents (or so called *schema* of the document) do not differ very much from application to application and that is why designated languages were defined to describe these requirements. DTD [33] or XML Schema [32] are the most common ones, the latter being a preferred choice where more thorough description or type awareness is desired. The abbreviation XSD (XML Schema Definition) is used for concrete schema definition (instance) in XML Schema language.

Having a separate language for XML schema definition is useful for several reasons:

- It allows using generic parsers and validators to be created. These validators can validate any document against any schema.
- Knowledge of the schema of the document can be used to optimise the strategy for storing and querying documents with query (XPath [35], XQuery [36]) and update languages (XQuery Update Facility [37]).
- One part of the XML Schema language is the definition of built-in data types and a set of rules to create user-defined types. This type system is used in other XML-related standards e.g. XPath or XSLT [15].

1.3 Aim of the Thesis

When required schema evolution is performed, document revalidation may be the harder part of the task. The naïve approach to document revalidation would be using a generic XML schema validator and checking each document one by one.

This approach can be considerably time-consuming, and furthermore many manual corrections can be performed automatically thanks to knowing the nature of the changes and the fact that the document was valid against the old schema. Besides, many changes may not require any revalidation at all.

The aim of this thesis is to design an algorithm that will be able to

- detect the set of changes between two versions of a schema,
- categorize the detected changes,
- decide, whether documents valid against the old schema stay valid against the new schema or whether they have to be revalidated,
- if the documents have to be revalidated, suggest a transformation stylesheet or script that will revalidate the documents.

1.4 Structure of the Thesis

The thesis is organized as follows. In Chapter 2, we examine existing approD-BLP:conf/dexa/2005aches and systems for XML schema evolution. In Chapter 3 we describe the XSem framework and its constructs for modeling platform-independent and platform-specific diagrams. In Chapter 4 we show how to identify differences between two versions of a schema. Chapter 5 contains formal

definitions of all types of differences that can be detected between two versions of a schema. Chapter 6 describes an algorithm that produces a revalidation script. Chapter 7 briefly describes the current state of the implementation and DBLP:conf/dexa/2005describes experiments with the real-world schemas. Finally, Chapter 8 concludes and provides future research directions.

Chapter 2

Related Work

In this chapter existing approaches to XML schema evolution will be described and categorized.

One way to categorize is based on the level where the changes are made by the designer and detected by the algorithm as suggested by Mlynkova and Necasky [21]. Existing approaches can be then classified into three groups according to whether changes are executed and detected:

- in the XML schemas (written in one of the XML schema languages), so-called *logical* level. At the logical level changes are detected directly in the evolved schema (in one of the XML schema languages, typically DTD or XML Schema), they are analyzed and the algorithm decides, whether it is necessary to revalidate the documents.

- in the diagrams visualizing the XML schema (diagrams directly visualizing constructs of a specific XML schema language). Some kind of visualization of the schema is used in order to simplify orientation in the schema and to avoid errors in schema definitions written manually. Changes are detected in the visualization.

- in a UML diagram used to model an XML format (usually annotated using comments or stereotypes to specify the translation to an XML schema). Changes are detected in the UML diagram and in the applications of the stereotypes and annotations on the UML constructs.

2.1 X-Evolution

X-Evolution proposed by G. Guerrini and M. Mesiti in [12] is an example of a system built upon graphical editor for creating schemas in the XML Schema language [32].

2.1.1 Evolution Primitives

At first the authors defined a set of *evolution primitives* divided into three categories – *insertion*, *modification* and *deletion* and three contexts where these primitives can be executed – *simple type*, *complex type* and *element*.

	Insertion	Modification	Deletion
Simple Type	insert_glob_simple_type* insert_new_member_type*	change_restriction change_base_type rename_type* change_member_type global_to_local* local_to_global*	remove_type* remove_member_type*
Complex Type	insert_glob_complex_type* insert_local_elem° insert_ref_elem° insert_operator°	rename_local_elem rename_type* change_type_local_elem° change_cardinality° change_operator° global_to_local* local_to_global*	remove_element° remove_operator° remove_substructure° remove_type*
Element	insert_glob_elem	rename_glob_elem* change_type_glob_elem ref_to_local* local_to_ref*	remove_glob_elem*

Table 2.1: Evolution Primitives within X-Evolution

The primitives are categorized as per their effect on the documents. Primitives marked with * in Table 2.1 do not alter validity of documents, those marked with ° operate on the structure of an element.

The set of all primitives is denoted $\mathcal{P}$, the set of all primitives not altering validity of documents is denoted $\mathcal{P}^*$, the set of XML schemas is denoted $\mathcal{SX}$ and the set of XML documents is denoted $\mathcal{DOC}$.

Some primitives are also associated with *applicability conditions* - conditions that guarantee consistency of the updated schema (e.g. referenced type can not be deleted).

X-Evolution system provides the user with a visualization of an XML Schema where he/she can select a construct to evolve. Based upon the selected construct, he/she is given a set of available primitives from which he/she can choose. Before the evolving operation is executed, the user may specify some additional arguments based on the type of the primitive.

2.1.2 Validation

To achieve document re-validation, the authors introduce a recursive function *validS*, which checks, whether a sequence of elements adheres to a type structure t (prescribed subelements). Relation $t_1 \sqsubseteq_{ST} t_2$ is introduced for type structures symbolizing that the legal values of t_1 are known to be contained among the legal values of t_2.

Algorithm 1 Revalidate

Input: $p \in \mathcal{P}, d \in \mathcal{DOC}, sx \in \mathcal{SX}$
Output: true $\Leftrightarrow$ d is valid for the updated schema

1: **if** $(p \in \mathcal{P}^*)$ **then**
2: **return** true
3: **else if** $(p \in \{rename\_glob/local\_elem\})$ **then**
4: let l be the element tag to remove/rename
5: **return** $(getElems(getPaths(l, sx), d) = \emptyset)$
6: **else if** $(p = change\_type\_glob/local\_elem(l, \tau_N, sx) \wedge \tau_N \in \mathcal{CT})$ **then**
7: let t_N be the structure of τ_N
8: **return** $(\forall e \in getElems(getPaths(l, sx), d) :$
$validS(children(e), init(t_N), t_N))$
9: **else if** $(p \in \{change\_restrict, change\_base/member\_type,$
$change\_type\_glob/local\_elem\})$ **then**
10: let τ_O be the old simple type and τ_N the updated one
11: **if** $(\tau_O \sqsubseteq \tau_N)$ **then**
12: **return** true
13: **end if**
14: **return** $(\forall e \in getElems(getPaths(\tau_O, sx), d) : content(e) \in \|\tau_N\|)$
15: **end if**

The *Revalidate* algorithm (see Algorithm 1) takes as an input an evolution primitive $p \in \mathcal{P}$, XML schema $sx \in \mathcal{SX}$ and XML document $d \in \mathcal{DOC}$ valid against sx and returns true when document d remains valid after applying p on sx.

Thanks to the categorization of primitives the algorithm is able to determine whether revalidation of XML documents is needed after the schema evolution by examining the type of the primitive p. In the case when $p \in \mathcal{P}^*$, the XML document d itself does not need to be examined at all. In other cases the algorithm must continue with examining the document d, but only the affected part of the document needs to be revalidated, not the document as a whole.

Function *getElems*$(\mathcal{E}, d)$ evaluates a set of XPath expressions $\mathcal{E}$ of certain format (containing only child axis steps) on document d and returns the corresponding elements. Function *getPaths*(e, t) returns an XPath expression consisting of steps along the child axis from the root of the schema, passing through e, reaching an element of type, structure, or tag t. Function *content*(e) returns value of XML attribute/element with simple content e.

2.1.3 Document Adaptation

In the next step the authors introduce the function *adaptS*, which is an extension to *validS*. This function alters the list of subelements that is not valid for the type structure. Parameter *opt* controls whether the list is altered by adding and/or removing new subelements.

When a new subelement is added by *adaptS*, the minimal structure for the subelement is created (minimal XML fragment conforming to the desired structure with default values for data content elements - leaves in XML tree).

The algorithm *Adapt* utilizes the *adaptS* function which takes as an input a primitive $p \in \mathcal{P}$, $d \in \mathcal{DOC}$ and $sx \in \mathcal{SX}$. The result is a document d' valid against sx_N (sx_N being the XML Schema obtained by application of p on sx).

Authors conclude with two propositions:

$$valid(d, sx_N) \equiv revalidate(p, d, sx)$$
$$valid(adapt(p, d, sx), sx_N) = \textbf{true}$$

In another article [11] G. Guerrini and M. Mesiti describe *XSchemaUpdate language*[1]. XSchemaUpdate statements follow the pattern:

UPDATE SCHEMA ObjectSpec
UpdateSpec
AdaptSpec?

[1] not to be confused with XQuery Update Facility language [37]

ObjectSpec being the set of nodes in an XML Schema to be updated, *UpdateSpec* the update operation and *AdaptSpec* the optional document adaptation statement performed for each document valid against the evolved schema.

With document adaptation statement new content can be created with non-default values as opposed to content created by *adaptS* function only.

2.1.4 Discussion

X-Evolution system provides the user with a narrow set of evolution primitives that can be performed upon XML schemas of a certain format. XML documents are restricted to elements and subelements; without attributes. Similarly, the XML schemas can contain only basic constructs of XML Schema language: simple and complex types and operators `<xs:choice>`, `<xs:sequence>` and `<xs:all>`.

There are no links to the conceptual model which could be utilized when new content needs to be added during document revalidation; thus it is up to the user to write the adaptation clauses or update the documents after revalidation.

Some very common real-world evolution operations are not considered in $\mathcal{P}$, e.g. moving content, adding a wrapping element for elements or transforming attributes to subelements and vice versa.

On the other hand, some primitives concerning purely technical changes are added into the set of primitives $\mathcal{P}$ (e.g. changing global type to local and vice versa or referenced element to local and vice versa).

All algorithms and operations expect a single evolution primitive $p \in \mathcal{P}$ as an input. Only one change is supported in each evolution cycle and evolution with multiple changes is not elaborated.

2.2 XEM: XML Evolution Management

XML Evolution Management, XEM [13] is an approach to manage schema evolution where DTD is used as a schema language. It deals both with changes in DTD and XML documents (*instance documents*).

2.2.1 Evolution Primitives

The proposed primitives are divided into two categories:

- changes in DTD
- changes in instance documents

Both DTD and instance XML documents are represented in the system as directed acyclic graphs and the primitives are defined as operations on these graphs. Table 2.2 lists primitives from both groups.

Each primitive comes with a description of its semantics and a set of preconditions, that must be satisfied before the primitive can be executed. The effect of each primitive on the DTD and instance documents is described.

It is proven that the set of DTD primitives is complete meaning any possible change in DTD can be expressed as a sequence of application of the primitives. This is done by defining four operations on a DTD graph and finding equivalents for these operations in the set of change primitives. Table 2.3 shows the operations together with their equivalents.

It is apparent that any DTD can be created from an empty DTD solely by executing *create-ver* and *add-edge* operations. Likewise, any DTD can be reduced to an empty DTD solely by executing *delete-ver* and *remove-edge*. Combining the previous statements, given two arbitrary DTD graphs G and G', there is a finite sequence of operations listed in Table 2.2 that transforms G to G'. Thus the set of evolution primitives listed in Table 2.2 is complete.

2.2.2 Implementation

XEM-Tool, which implements XEM, uses object-oriented mapping for manipulating DTDs and instance XML documents. The input DTD is processed by *DTD Manager* component which builds a DTD graph and then generates Java classes (ELEMENT definitions are translated to classes, attribute definitions to properties, parent-child relationships are stored in a *children* vector of each class).

#	DTD Operation	Description
1	createDTDElement(s t)	Create target element type with name s and content type t
2	destroyDTDElement()	Destroy target element type
3	insertDTDElement(e, i, q, v)	Add element type e with quantifier q and default value v at DTD position i to target element type
4	removeContentParticle(i)	Remove content particle at DTD position i in target element type
5	changeQuant(i, q)	Change quantifier of content particle at DTD position i in target element type to q
6	convertToGroup($start$, end, t)	Group content particles from DTD position *start* to *end* in target element type into a group of type t
7	flattenGroup(i)	Flatten group at DTD position i in target element type
8	addDTDAttr(s, t, d, v)	Add attribute type with name s with type t, default type d, and default value v to target element type
9	destroyDTDAttr(s)	Destroy attribute type with name s from target element type
#	**XML Data Operation**	**Description**
10	createDataElement(e, v)	Create target element node with type e and value v
11	addDataElement(e, i)	Add element node e at position i in target element node
12	destroyDataElement()	Destroy target element node
13	addDataAttr(s, v)	Add an attribute with name s and value v to target element node
14	destroyDataAttr(s)	Destroy attribute with name s in target element node

Table 2.2: Evolution Primitives within XEM

Operation	**Description**	**Taxonomy Equivalent** (see Table 4.2)
create-ver	Creates new dangling vertex	1, 6, 8
add-edge	Adds an edge between two vertices	3, 6, 8
delete-ver	Deletes vertex with zero out-degree and removes all incoming edges	2, 7, 9
remove-edge	Removes the edge between two vertices	4, 7, 9

Table 2.3: The DTD Graph Operations and Their Equivalent XEM Primitives

Instance XML documents are processed by an *XML Document Manager* component, that converts XML data to objects - instances of classes generated by *DTD Manager*.

When a DTD changing evolution primitive is executed in the system (e.g. addDTDAttr), some of the classes generated by *DTD Manager* need to be regenerated (this includes creating and compiling new Java source files, creating instances of the new classes, copying the existing contents from the instances of the old classes in memory and removing the old classes and source files).

2.2.3 Discussion

XEM deals with DTD which is much simpler than XML Schema or the XSem-H model used in this thesis (3). The set of proposed primitives is proven to be sound and complete in the terms of being able to transform any DTD to any other DTD, but for a large portion of common evolution changes this often leads to removing a significant part of the XML document and recreating it again. E.g. when an element is renamed, it must be removed and then added under new name; if the root element is removed, the whole XML document is first deleted and its structure recreated again. The same holds for moving content. In this process, the structure is created properly by the algorithm, but the data is lost.

Again, as in the case of *X-Evolution*, the conceptual model is not considered and thus cannot be utilized when new content is added in the instance documents. While *X-Evolution* provides the option of *Adapt clauses* for altering the content after the evolution primitive is executed, *XEM* allows only specifying default values for newly created content.

2.3 CoDEX

CoDEX, Conceptual Design and Evolution of XML schemas [17], is an example of an approach to schema evolution built upon a conceptual model.

The approach is implemented in *CoDEX-tool* software. It enables a user to design a conceptual model for an XML schema (and also the conceptual model can be created for already existing schemas following some design rules) and conduct the evolution changes in the conceptual model. The changes made in

the model are logged and when the evolution process is finished, the resulting changes are performed in the XML schema. Documents associated with the evolved schema are then updated according to the XML schema evolution steps.

2.3.1 Conceptual Model

The conceptual model is a graph with nodes of four classes - *elements*, *types*, *groups* and *modules* together called *basic components*. In addition, Each component has in addition a set of key-value pairs of *properties*.

Conceptual model can be translated to an XSD, where elements are mapped to `<xs:element>` constructs, types to `<xs:complexType>` and `<xs:simpleType>` and groups to `<xs:all>`, `<xs:sequence>`, `<xs:choice>` respectively; using the *Venetian blind design pattern* [8] (which is also required for importing existing schemas).

2.3.2 Schema Evolution

The evolution process is conducted by the user via a graphical interface in the conceptual model. Each user's single action (called *design step* by the authors, in previously described system these were called *evolution primitives*) is recorded by the logging component.

When the user is satisfied with the new version of the model the recorded design steps are minimized and normalized. This is due to the design process conducted by the user often not being straightforward. When design steps are grouped according to the point in the schema where they were conducted, some groups of design steps may exhibit mutually annulling (adding and removing the same element), other groups of design steps can be reduced (adding and renaming an element can be replaced by adding the element with the final name).

There are 53 simple rules for combining and reducing design steps. The following rule combining creating and renaming an element can serve as an example:

$$create\_element(id, name, content) + rename\_element(id, name, name') \\ \rightarrow create\_element(id, name', content)$$

The minimized set of design steps is translated into XML evolution steps through which the original schema is evolved to the evolved schema.

In such a case where the documents valid against the original schema are no longer valid for the the evolved schema XML update operations are generated.

2.3.3 Discussion

In contrast to previous approaches, CoDEX allows the user to make a quantity of changes in each evolution cycle (both XEM and X-Evolution allowing only a single primitive).

Process of handling existing XML schemas is also elaborated in this approach.

In general, the proposed conceptual model is closer to a visualization of an XML Schema language than to a platform-independent model. The lack of conceptual model is directly responsible for a problem mentioned in [17] (Limits of conceptual schema evolution). An example is given where the user moves element `address` from element `owner` to element `producer`. The structure of both the elements `address` is the same (they would probably refer to the same type), but semantically the meaning is different (the address of the producer is not the same as the address of the owner) and moving the contents of `address` from element `owner` to element `producer` would create valid, yet semantically incorrect document. This problem is declared to be in-built and detecting these changes unsolvable without user interaction. In the subsequent text we will show that the approach proposed in this thesis can handle these situation satisfyingly.

2.4 Evolution using UML Class Diagrams

The Unified Modeling Language [26, 27], UML, is nowadays widely used for platform independent modeling of system infrastructure and data. XML Schema derivation from UML class diagrams is supported by commercial tools [29]. The schema derivation can be fully automatic or adjusted by applying stereotypes provided in a UML profile.

Adding support for schema evolution is proposed in [10]. The authors utilize UML class diagrams for the conceptual model. Mapping between building blocks of class diagrams and XML Schema constructs is described in the Table 2.4, the translation is fully automatic.

UML block	XML item(s)
class	element, complex type with ID attribute and key
attribute	subelement of the corresponding class complex type
association	reference element, with IDREF attribute referencing the associated class and keyref for type safety (key/keyref references)
generalization	complex type of the subclass is defined as an extension of the complex type of the superclass

Table 2.4: Mapping Between UML and XML Schema Constructs

Output of the translation operation is the XSD and also a set of *translation rules* that store correspondence between UML and XML constructs.

The evolution operations are conducted upon the class diagram instead upon the (visualized) XML schema. The set of evolution primitives consists of basic operations for working with UML diagrams (i.e. addition or deletion of classes, attributes and associations).

The propagation of changes proceeds in several steps. First, the *translation rules* keeping correspondence between UML and XML constructs are updated to reflect the changes. Deduced changes in the set of translation rules serve as an input for the next step – propagation of changes to the XML schema and XML documents.

Each change triggers one or more *XSL transformations* [16], both for XML schema and XML documents.

2.4.1 Discussion

UML class diagrams are very suitable for platform independent modeling. The authors decided to provide a straightforward translation of the UML diagram without any means to adjust and tailor the shape of the resulting XSD.

Potential necessity in some larger systems of having more schemas derived from one conceptual diagram is not considered.

Chapter 3

XSem

3.1 Conceptual Modeling

Conceptual modeling is a top-down approach to system and data design, which concentrates on correct depicting the problem domain - defining the particular concepts comprising the system and relationships between these concepts.

Conceptual model emerges from user requirements analysis and is defined using a modeling language sufficiently universal and independent of the implementation language. It serves as an interconnecting foundation for the individual components of the system (and these components can be implemented using various technologies and languages).

In the implementation phase of the system development, concepts from the conceptual model serve as drafts for corresponding constructs in the respective implementation language (e.g. classes in object oriented programming or tables in relational database design). Some design tools provide features of automatic generation of the implementation constructs from the conceptual model.

Conceptual modeling language should be also reasonably comprehensible to the customer and serve to ease the clarification and specification of the user requirements during each iteration of system specification.

The most frequent conceptual models in use are the Unified Modeling Language (UML) [27] [26] class diagrams and the Entity Relationship model [31].

3.2 XSEM

XSem is an approach to modeling XML data using *Model Driven Architecture* (MDA) [20]. A prototype implementation of XSem in a CASE[1] tool called XCase [7] is available.

The model has two interconnected layers – platform-independent model (PIM) which utilizes UML class diagrams and platform-specific model (PSM) which again utilizes UML class diagrams but extended with *XSem profile*.

The reason for having two layers for modeling is to separate the layers of system design. Platform-independent layer is used to thoroughly and completely describe the problem domain without redundancies. PIM can be then used to design the data storage and provides a complete definition of each concept.

Different parts of the problem domain are then used by individual components of the systems, usually each component does not work with all the concepts, but only with a subset. On the other hand, some concepts are used by several (or all) system components.

If we move to XML data modeling, each component may have individual requirements on the portion of the used data and their structure and layout in the XML document – XML format. Although the XML formats may differ, they all concern the same problem domain and reference the concepts described at the platform-independent level, but each format contains different subset of the concepts and different parts of the concepts. Each XML format is a kind of an *XML view* over the data.

For example, the component of the system that matches accepted payments to the orders in the system will only need in its incoming message the identifier of the order and the amount of money paid to locate the order and verify the price, whereas the component that realizes the shipping will need the list of items in the order and the identifier of the customer who issued the order. See platform-independent diagram in Figure 3.1 and two derived platform-specific diagrams in Figure 3.2.

[1] Computer-aided software engineering

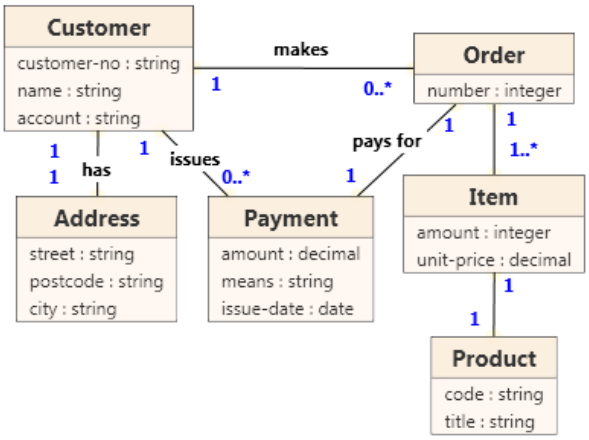
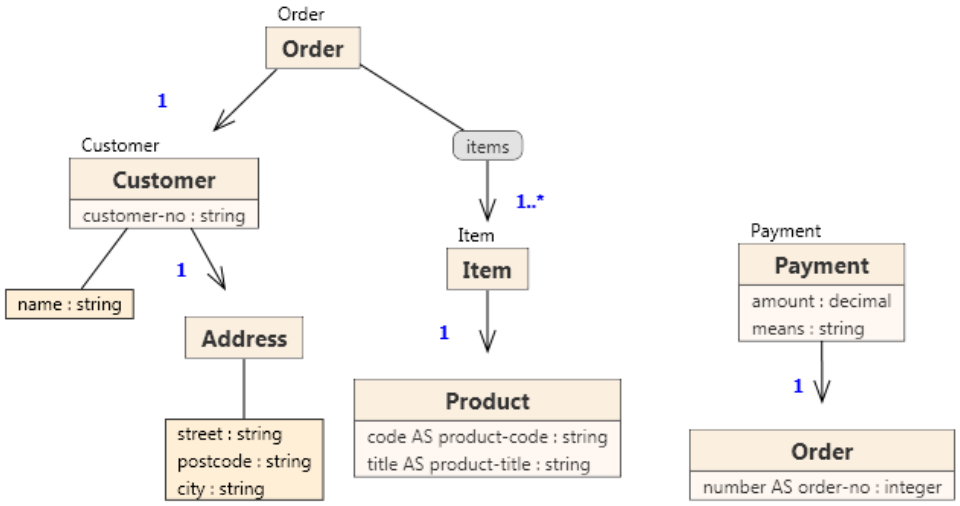


Figure 3.1: Simple Platform-Independent Diagram



(a) Order Diagram (b) Payment Diagram

Figure 3.2: Platform-Specific Diagrams

As a result, a complex system will contain many XML formats where various concepts appear repeatedly in different forms and structure. Changes in one XML format may cause necessary changes in other formats to keep the model consistent.

That is why XSem uses two-layer model with the two interconnected layers. The platform-specific model is comprised of several diagrams, each being a model

for one specific XML format.

Changes at the specific layer are verified against the independent layer and vice-versa. The user is offered with propagation of changes from one layer to the other one or notified that the action can not be performed, because it would put the model in an inconsistent state (e.g. deleting a concept at the independent level can not be accomplished unless all references of the concept at the specific level are removed).

With the two layer system, the user is provided with sufficient level of freedom when designing the individual XML formats, but is prevented from making changes that would render the model inconsistent.

3.3 Platform-Independent Model – UML Class Diagrams

UML class diagrams are part of the Unified Modeling Language. They are used to model classes in object-oriented programming and also for data modeling. The following class diagram constructs are used in PIM[2]:

- *classes*[3] (the set of all PIM classes will be denoted $\mathcal{C}_{pim}$)
- *class attributes* (the set of all PIM attributes will be denoted $\mathcal{A}tt_{pim}$)
- *types* (the set of all types will be denoted $\mathcal{T}$)
- *association ends* (the set of all association ends will be denoted E_{pim})
- *associations* (the set of all PIM associations will be denoted $\mathcal{A}_{pim}$)

Definition 3.3.1 (PIM Model)**.** *A* PIM Model *is a union:*

$$\mathcal{M}_{pim} = \mathcal{C}_{pim} \cup \mathcal{A}tt_{pim} \cup \mathcal{T} \cup E_{pim} \cup \mathcal{A}_{pim}$$

Figure 3.3 contains a metamodel of PIM constructs. A construct referencing another construct is shown as an association between the two.

[2]PIM model can be further extended with additional constructs, namely *association classes* and *generalizations*, which are both available in *XCase*.

[3]When necessary, we will use notions *PIM classes* and *PSM classes* to distinguish constructs on PIM and PSM level (and similarly for other constructs).

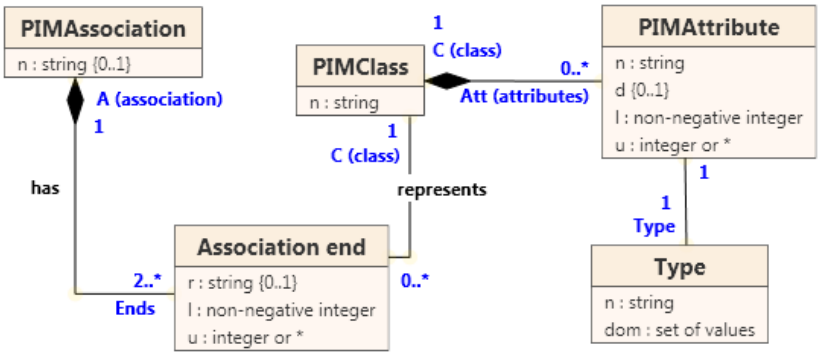


Figure 3.3: PIM Metamodel

In the following text we also assume a finite set of finite-length strings $\mathcal{L}$.

Class An instance of a class construct is intended to model one particular kind of real-world objects that all share the same features and semantics. In PIM diagrams, classes are depicted as boxes with their name in bold-face and list of attributes below. Diagram in Figure 3.1 contains six classes: `Customer`, `Address`, `Payment`, `Order`, `Item` and `Product`.

Definition 3.3.2 (Class)**.** *A* class $C \in \mathcal{C}_{pim}$ *is a 2-tuple:*

$$C = (n, \mathcal{A}tt)$$

where:

- $n \in \mathcal{L}$ *is a string called* class name
- $\mathcal{A}tt \subseteq \mathcal{A}tt_{pim}$ *is a list of attributes of* C*,* $\mathcal{A}tt$ *can be empty.*

Class attributes Class attributes model characteristics, features and properties of classes. A *data type* is assigned to each attribute and each instance of the attribute is assigned with a value belonging to the type's *domain*. In PIM diagrams attributes are listed inside the class box. Diagram in Figure 3.1 contains attributes `customer-no`, `name`, `account` in class `Customer`, `number` in class `Order` etc.

Definition 3.3.3 (Attribute)**.** *An* attribute $a \in \mathcal{A}tt_{pim}$ *is a 6-tuple:*

$$a = (C, n, \mathit{Type}, d, l, u)$$

where:

- $C \in \mathcal{C}_{pim}$ *is a class to which this attribute belongs to*
- $n \in \mathcal{L}$ *is a string called* attribute name
- $Type \in \mathcal{T}$ *is the* type *assigned to the attribute*
- $d \in Type.dom_{Type}$ *is the* default value *assigned to the new instances of* a*;* d *can be omitted – in that case new instances of* a *are assigned the default value defined for type* $Type$
- $l \in \mathbb{N}_0$ *denotes* lower cardinality *of the attribute* a
- $u \in \mathbb{N} \cup \{*\}$ *denotes* upper cardinality *of the attribute* a*, value* $*$ *denotes unlimited upper cardinality*

and following conditions must be satisfied:

$$\begin{aligned} \forall a \in \mathcal{A}tt_{pim} : \; & a.l \leq a.u \; \wedge \\ & \exists K \in \mathcal{C}_{pim} : a \in K.\mathcal{A}tt \; \wedge \\ & a.C = K \leftrightarrow a \in K.\mathcal{A}tt \end{aligned}$$

Type A type can be interpreted as a named set of possible values. Types are usually firmly defined in the implementation languages (with a possibility to derive new types from the built-in ones). It is thus problematic to provide a set of types at the platform-independent level. Either the platform-independent types must be some generic types that can be translated into each implementation language (e.g. platform-independent type `string` can be translated into `xs:string` when XML Schema is the implementation language and into `varchar` when the implementation language is SQL).

The other solution is to adopt a type system of a selected implementation language at the platform-independent level. Since we are concerned with XML data evolution, we will adopt a type system of the XML Schema language (see [28]).

The above mentioned specification defines 19 built-in types (see Figure 3.4) and rules for deriving user-defined types.

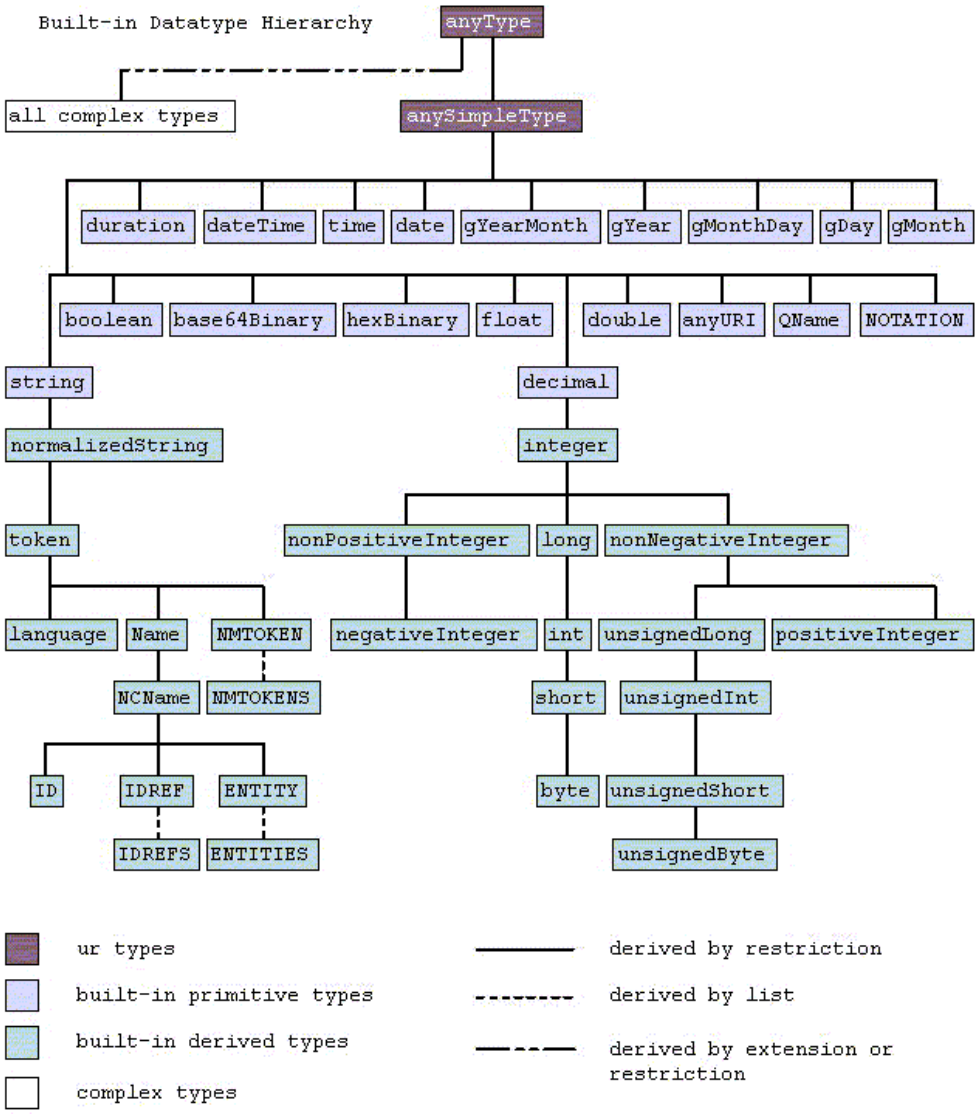


Figure 3.4: XML Schema Built-in Datatype Hierarchy (taken from [28])

Definition 3.3.4 (Type)**.** *A* type T *is defined as:*

$$T = (n, dom_T)$$

where:

- $n \in \mathcal{L}$ *is the* name *of data type* T

- *dom_T is a set of all allowed values for type T called* domain *of type T. Domains of built-in types are firmly specified and domains of user derived types are defined by the rules of the type system.*

Association Figure 3.5 contains another example of a PIM diagram that demonstrates associations and association ends.

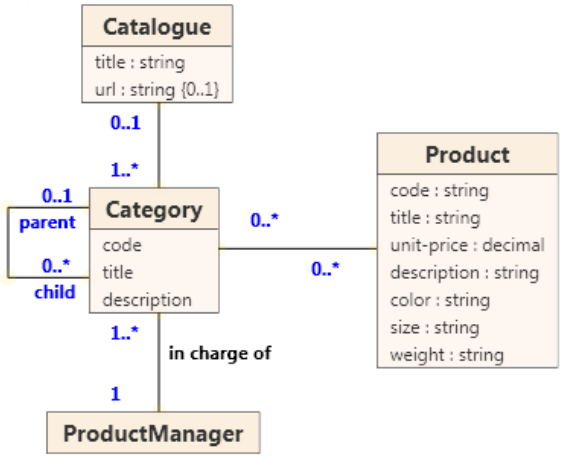


Figure 3.5: Example of a Simple PIM Diagram

Associations model relations between real-word concepts. An association can connect an arbitrary amount of classes, each class then participates in the association with certain role and cardinality. In PIM diagrams associations are depicted as lines connecting classes. One class can participate in one association under multiple roles. Class `Category` in Figure 3.5 participates under two roles – `parent` and `child` with multiplicities (0, 1) and (0..*) respectively – in an association (each category can have an arbitrary amount of child categories, and each child category has one parent category). Auxiliary construct association end models the role and cardinality of each participation of a class in an association.

Definition 3.3.5 (Association)**.** *An* association $A \in \mathcal{A}_{pim}$ *is defined as:*

$$A = (n, Ends)$$

where:

- $n \in \mathcal{L}$ *is the* name *of the association; n is optional.*

- *Ends* $\subseteq E_{pim}$ *is a list of association* ends.

Association end is an auxiliary construct modeling participation of the class in the association.

Definition 3.3.6 (Association End)**.** *An* association end $E \in E_{pim}$ *is defined as:*

$$E = (A, C, r, l, u)$$

where:

- $A \in \mathcal{A}_{pim}$ *is an* association *of which* E *is part of*
- $C \in \mathcal{C}_{pim}$ *is a* class *to which this association end belongs to*
- $r \in \mathcal{L}$ *is the* role *of* C *in the association* A*;* r *can be omitted.*
- $l \in \mathbb{N}_0$ *denotes* lower cardinality *of the participation of* C *in* A
- $u \in \mathbb{N} \cup \{*\}$ *denotes* upper cardinality *of the participation of* C *in* A*, value* $*$ *denotes unlimited upper cardinality*

3.4 Platform-Specific Model

For the purposes of modeling XML data, UML class diagrams were extended with a profile named *XSem* containing one stereotype for *Association* and *Property* constructs and several stereotypes for *Class* construct. This extension to original UML class diagrams named XSem-H is used as the platform-specific model.

A visual notation for PSM diagrams was created to transparently display models of XML data with emphasis on the two significant features of XML data model – hierarchical character and nesting.

XSem-H diagrams contain following constructs:

- *PSM classes* (the set of all PSM classes will be denoted $\mathcal{C}_{psm}$)
- *PSM attributes* (the set of all PSM attributes will be denoted $\mathcal{A}tt_{psm}$)
- *PSM associations* (the set of all PSM associations will be denoted $\mathcal{A}_{psm}$)
- *attribute containers* (the set of all attribute containers will be denoted $\mathcal{AC}$)
- *content containers* (the set of all content containers will be denoted $\mathcal{CC}$)

- *class unions* (the set of all class unions will be denoted $\mathcal{CU}$)
- *content choices* (the set of all content choices will be denoted $\mathcal{CH}$)

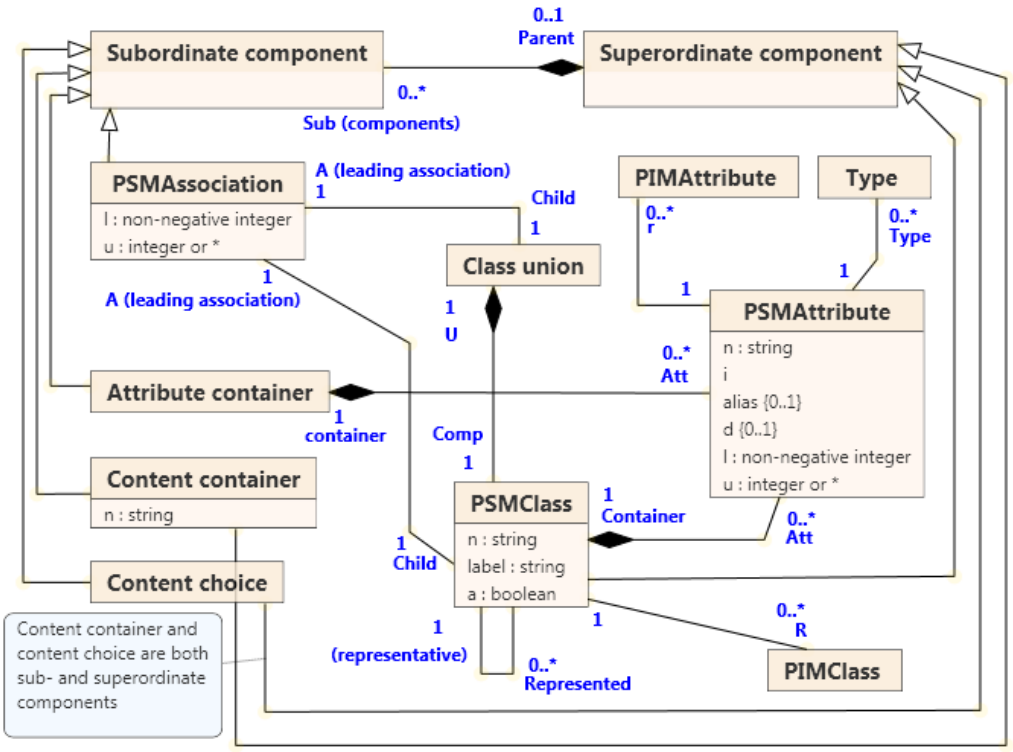


Figure 3.6: PSM Metamodel

Definition 3.4.1 (PSM Model)**.** PSM Model *is a union:*

$$\mathcal{M}_{psm} = \mathcal{C}_{psm} \cup \mathcal{A}tt_{psm} \cup \mathcal{A}_{psm} \cup \mathcal{AC} \cup \mathcal{CC} \cup \mathcal{CU} \cup \mathcal{CH}$$

Since XSem-H diagram is a tree (or a forest of trees), we will call PSM classes, attribute containers, content containers, class unions and content choices *nodes*.

For purposes of definition we further need two abstract constructs:

- *superordinate node* (the set of all superordinate nodes will be denoted $\mathcal{C}_{sup}$)

$$\mathcal{C}_{sup} = \mathcal{C}_{psm} \cup \ \mathcal{CH} \cup \ \mathcal{CC}$$

- *subordinate node* (the set of all subordinate nodes will be denoted $\mathcal{C}_{sub}$)

$$\mathcal{C}_{sub} = \mathcal{A}_{psm} \cup \mathcal{CH} \cup \mathcal{CC} \cup \mathcal{AC}$$

Figure 3.6 contains a metamodel of PSM constructs. The diagram also shows three PIM constructs (`PIMClass`, `PIMAttribute` and `Type`) that are referenced from PSM constructs.

Figure 3.7 contains an example of a PSM diagram using all types of constructs mentioned above.

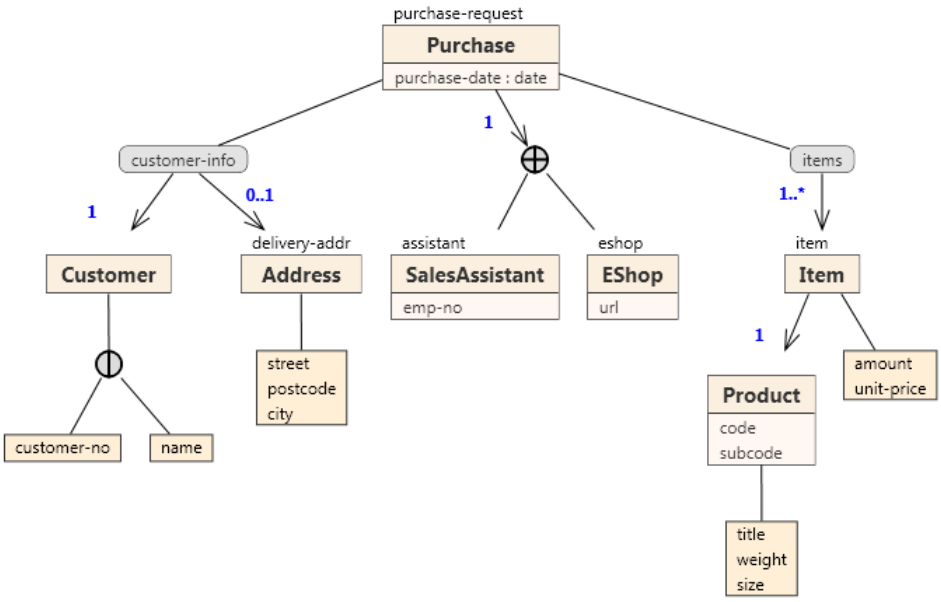


Figure 3.7: PSM Diagram Constructs

Significant Node

Superordinate Superordinate abstract construct is a node that can contain other constructs (subordinate constructs).

Definition 3.4.2 (Superordinate Node)**.** *A* superordinate *node* $P \in \mathcal{C}_{sup}$ *is defined as:*

$$P = (Sub)$$

where:

- $Sub \subseteq \mathcal{C}_{sub}$ *called* components *is an ordered sequence of subordinate constructs; Sub can be an empty sequence*

Subordinate A subordinate abstract construct is a type of construct that can occur among components of superordinate nodes.

Definition 3.4.3 (Subordinate Construct)**.** *A* Subordinate *construct* $S \in \mathcal{C}_{sub}$ *is defined as:*

$$S = (Parent)$$

where:

- $Parent \in \mathcal{C}_{sup}$ *is a superordinate node called* parent *of* S*; when* $Parent$ *is not specified (this will be indicated by a special value null, similarly with properties of other constructs), then* S *is one of the roots of the XSem-H forest*

The following invariant joins superordinate and subordinate constructs:

$$(\forall P \in \mathcal{C}_{sup})((\forall s \in P.Sub) \rightarrow s.Parent = P)$$

PSM Class A PSM class is the most important construct at the platform-specific level, because it brings the concepts modeled at the platform-independent level to the platform-specific level.

Each class is *derived* from exactly one PIM class and the link to the PIM class is maintained during the whole existence of the PSM class to ensure consistency between the two layers and allow mutual propagation of changes.

The diagram in Figure 3.1 contains one root PSM class `Purchase` and several other PSM classes (`Customer`, `Address` etc.).

Definition 3.4.4 (PSM Class)**.** *A* PSM class $C \in \mathcal{C}_{psm}$ *is a* superordinate *node*

$$C = (R, n, label, A, U, \mathcal{A}tt, Sub, Represented, a)$$

where:

- $R \in \mathcal{C}_{pim}$ *is the* represented class *of* C*, i.e. the PIM class which* C *was derived from*
- $n \in \mathcal{L}$ *is the* name *of* C*, it is initialized to* $R.n$ *but can be changed*
- $label \in \mathcal{L}$ *is the* element label *of* C*; label is optional*
- $A \in \mathcal{A}_{psm}$ *is the the* association *leading to* C*;* A *can be empty*
- $U \in \mathcal{CU}$ *is the the* class union *to which this* C *belongs;* U *can be empty*

- $\mathcal{A}tt \subset \mathcal{A}tt_{psm}$ *is an ordered sequence of PSM attributes of* C*;* $\mathcal{A}tt$ *can be empty*
- *Sub is an ordered sequence of components of* C*; Sub can be empty*
- $Represented \in \{P : \mathcal{C}_{psm} | P.R = C.R\}$ *is a link to the* represented PSM class*; it can be empty; when* $C.R$ *is not empty,* C *is called a* structural representative *of Represented. Notion* structural representative *is explained in paragraph Structural Representative in this section, p. 46.*
- $a \in (true, false)$ *is a boolean flag denoting that XML element modeled by* C *can contain any XML attribute, even when it is not defined in the model.*

The following invariants must hold ($\mathcal{D}$ *being an XSem-H diagram):*

$$\forall C \in \mathcal{C}_{psm} : C.A \neq null \rightarrow c.U = null \wedge C.A = null \rightarrow c.U \neq null$$
$$\forall C \in \mathcal{C}_{psm} \cap \mathcal{D} : C.A = null \wedge c.U = null \rightarrow C \in \mathcal{D}.roots$$

PSM class is depicted the similar way as PIM class with the element label above the class box. Components are arranged as children of the class node in the XSem tree (see class `Purchase` an its three components: content container `customer-info`, class union and content container `items` in Figure 3.7).

A PSM class construct models two different scenarios, depending on the value of *label*.

Classes with element labels model a subtree in an XML document. Let us denote E as the root XML element of the subtree modeled by C. The name of element E is equal to $C.label$, attributes of the class model XML attributes of E. Class `Eshop` in Figure 3.7 with an element label `"eshop"` and one attribute `"url"` thus models following XML subtree:

```
<eshop url="..."></eshop>
```

(The node is empty, because the class has no components)

If class C is a class without an element label, it models subelements and/or attributes of an element modeled by the first ancestor of C in the path from C to root of the diagram. For instance, class `Product` in Figure 3.7 has no element label. Its first ancestor that models an XML element is class `Item`. `Product` has two attributes (`code` and `subcode`) and a single component (attribute container with three attributes) which models the following fragment:

```
<title>...</title>
<weight>...</weight>
<weight>...</weight>
```

Both attributes and components of `Product` are inlined among the components and attributes of its parent `Item`. This principle is called *propagation*. Nodes that model XML elements (content containers and classes with element labels) will be called *significant nodes* and the first significant node in the path from a class C without element label to root will be called *closest significant node* to C. The content modeled by class C without element label is propagated to its closest significant node.

The whole subtree modeled by `Item` is then:

```
<item code="..." subcode="...">
    <amount>...</amount>
    <unit-price>...<unit-price>
    <title>...</title>
    <weight>...</weight>
    <size>...</size>
<item>
```

PSM Attribute In a similar way as a PSM class, also a PSM attribute is derived from a PIM construct - *PIM attribute*.

Definition 3.4.5 (PSM Attribute)**.** *A* PSM Attribute $a \in \mathcal{A}tt_{psm}$ *is defined as*

$$a = (R, C, \mathit{Container}, i, n, \mathit{alias}, \mathit{Type}, d, l, u)$$

where:

- $C \in \mathcal{C}_{psm}$ *is the PSM class to which this PSM attribute belongs*
- $R \in C.\mathcal{A}tt$ *is a PIM attribute of class* C *from which* a *is derived*
- $\mathit{Container} \in \mathcal{C}_{psm} \cup \mathcal{AC}$ *is the construct where* a *is placed; it is initialized to* $C \in \mathcal{C}_{psm}$ *but can be changed to an attribute container* $AC \in \mathcal{AC}$*. Placing attributes inside attribute containers is a way to adjust the form of modeled document.*
- $i \in \mathbb{N}_0$ *is the index of* a *in the collection* $a.\mathit{Container}.\mathcal{A}tt_{psm}$

- $n \in \mathcal{L}$ *is a string called* attribute name*; it is initialized to* $R.n$ *and cannot be changed*
- $alias \in \mathcal{L}$ *is an* alias *of this attribute, the name under which this attribute will appear in the modeled XML documents; alias can be omitted, in that case, value of* n *is used instead*
- $Type \in \mathcal{T}$ *is the* type *assigned to the attribute; it is initialized to* $R.Type$ *and cannot be changed*
- $d \in dom_{Type}$ *is the* default value *of* a
- $l \in \mathbb{N}_0$ *and* $u \in \mathbb{N} \cup \{*\}$ *denote* lower *and* upper cardinality

and the following conditions must be satisfied:

$$\begin{aligned} \forall a \in \mathcal{A}tt_{psm} : \; & a.l \leq a.u \; \wedge \\ & \exists K \in \mathcal{C}_{psm} \cup \; \mathcal{AC} : a \in K.\mathcal{A}tt \; \wedge \\ & a.Container = K \leftrightarrow a \in K.\mathcal{A}tt \end{aligned}$$

Meanings of d*,* l *and* u *are variants of the definitions for PIM* property *construct and their values are initialized to the values of the appropriate properties of* R*.*

In an XML document, a PSM attribute a models an attribute of the closest *significant node* when it is placed in a PSM class ($a.Container \in \mathcal{C}_{psm}$). A PSM attribute can be also placed inside an attribute container ($a.Container \in \mathcal{AC}$, see paragraph Attribute Container, p. 40), in that case it models an element with a simple content. See Figure 3.9 for an example of different usages of PSM attribute.

PSM Association PSM associations are derived from PIM associations, but unlike PSM classes and attributes, a PSM association does not represent a single PIM association. To give user the more freedom when creating the PSM diagram, a PSM association can be derived from a sequence of PIM associations. This enables the user to create a PSM association between two PSM classes whose respective PIM classes are not connected by a PIM association, but are connected through one or more other classes.

The PSM association is therefore linked to one or more PIM associations and the consistency must be maintained in the model (e.g. it is not possible to remove a PIM association which participates in a PSM association).

The exact rules for deriving a PIM association are described thoroughly in [24]. For the purposes of this work, the following simplified definition of a PSM association will suffice.

We will use notion $\mathcal{N}_{child} = \mathcal{C}_{psm} \cup \mathcal{CU}$. Nodes in $\mathcal{N}_{child}$ are the nodes that can participate in associations as target nodes.

Definition 3.4.6 (PSM Association)**.** *A* PSM Association $A \in \mathcal{A}_{psm}$ *is a* subordinate *construct defined as:*

$$A = (Parent, Child, l, u)$$

where:

- $Parent \in \mathcal{C}_{sup}$ *is the* parent *of A, i.e. the node where association A starts*
- $Child \in \mathcal{N}_{child}$ *is the second node participating in the association A, i.e. the node to which association A leads to (its target node)*
- l *and* u *denote* lower cardinality *and* upper cardinality *of the association, i.e. the interval of possible repetitions of Child*

Unlike PIM associations, PSM associations are always binary and each end can participate only once in the association.

PSM associations are always created associating two PSM classes ($\{Parent, Child\} \subset \mathcal{C}_{psm}$), but during the design process, association can be put into a superordinate container (either *content container* or *content choice*). So *Parent* can be changed to a content container or content choice construct (see Figure 3.7, where associations `Purchase-Customer` and `Purchase-Address` were put into a content container `customer-info`).

Associations can also be joined into the *class union*, so *Child* can be changed too (to a class union node) (see Figure 3.7, where associations `Purchase-SalesAssistant` and `Purchase-Eshop` were joined to a single association leading to a class union of classes `SalesAssistant` and `EShop`).

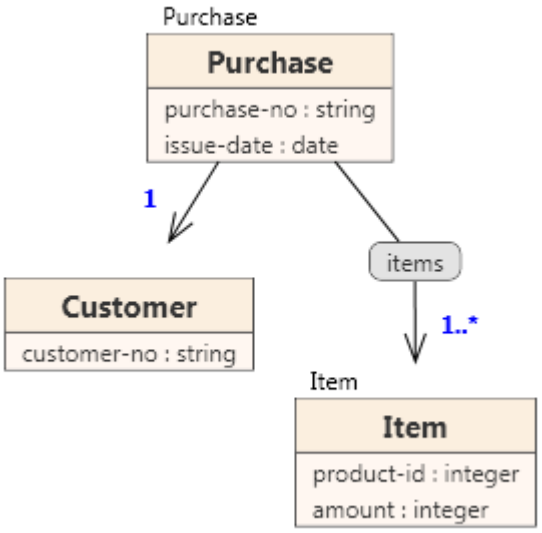
In XSem-H diagrams PSM associations are depicted as arrows and form edges in the XSem-H tree - *Parent* being the start node and *Child* being the target node.

In XML documents PSM association model nesting/propagation:

- when $A.Child$ is a class with an element label, subtree modeled by this class will be a subtree of an element modeled by the closest significant node to $A.Child$

- when *A.Child* is a class without an element label, content modeled by *A.Child* is inlined in the content modeled by the closest significant node to *A.Child*
- when *A.Child* is a class union, the principle passes to its child nodes (see paragraph Class Union in this section, p. 44)

Figure 3.8 contains two examples of PSM associations. The one between classes `Purchase` and `Customer` leads to a class without label, so the child node is propagated upwards. The other one leads to a class with element label, so new element (`Item`) must exist for each occurrence of the child node. The second association was also moved to the content container `items` which wraps the child nodes in the XML element `items`.



(a) Example for usages of PSM associations.

```
<Purchase purchase-no="###"
          issue-date=###"
          customer-no="###">
  <items>
    <Item amount="#"
          product-id="#" />
    <Item amount="#"
          product-id="#" />
    <Item amount="#"
          product-id="#" />
  </items>
</Purchase>
```

(b) Corresponding XML fragment

Figure 3.8: PSM Associations and a Corresponding XML Fragment

Attribute container An *Attribute container* is one of the constructs created in order to give the user a wider set of options when turning problem domain concepts into the model of XML documents. *Content container* is another example of such a construct.

Attribute container is convenient when the demand is to have attributes of classes in the form of XML elements (with simple content) instead of XML attributes.

Although the choice between the two possible representations of PSM attributes – XML elements or XML attributes – can in some cases be just a matter of the user's preference, when $a.u > 1$ for an attribute a, the form of XML element is the only correct choice (since XML data model does not allow multiple occurrence of the same attribute in one node).

Definition 3.4.7 (Attribute Container)**.** *An* attribute container $AC \in \mathcal{AC}$ *is a* subordinate *node*

$$AC = (Parent, \mathcal{A}tt)$$

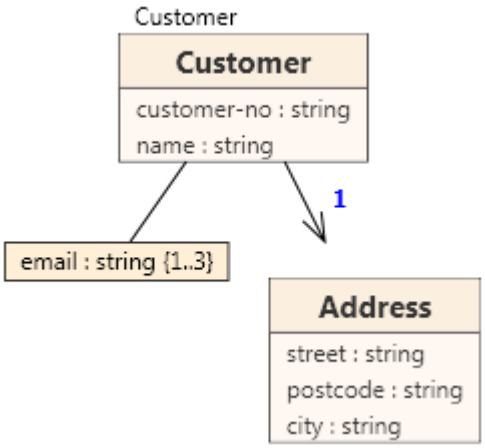
where:

- $Parent \in \mathcal{C}_{sup}$ *is the* parent *of* AC
- $\mathcal{A}tt \subseteq \mathcal{A}tt_{psm}$ *is an ordered sequence of PSM attributes of* AC*;* $\mathcal{A}tt$ *can be empty*

And the following condition must hold

$$\forall AC \in \mathcal{AC}, a \in \mathcal{A}tt_{psm} : a.Container = AC \leftrightarrow a \in AC.\mathcal{A}tt$$

In XSem-H diagrams, attribute containers are depicted as rectangles with the list of attributes inside. Each attribute in $AC.\mathcal{A}tt$ models an XML element with simple content in the content of the closest *significant node* of AC (with correct cardinality). Figure 3.9 shows different usages of PSM attributes and an attribute container. In this example, attributes of both classes `Customer` and `Address` model XML attributes of XML element `Customer`, because class `Address` does not have the element label and, therefore, its attributes are propagated to the closest *significant node*. Attribute `email` is placed in an attribute container, that is why it models XML elements `email`, which can appear 1-3 times inside the content of element `Customer`.



(a) Example for usages of PSM attributes.

```
<Customer customer-no="###"
          name="###"
          street="###"
          postcode="###"
          city="###">
  <email>###</email>
  <email>###</email>
</Customer>
```

(b) Corresponding XML fragment

Figure 3.9: PSM Attributes, an Attribute Container and a Corresponding XML Fragment

Content Container A *content container* is a simple construct for modeling of wrapping elements; i.e. elements that are not modeled at the platform-independent level. In other words, they are not a representation of a concept.

One of the convenient usages is to wrap cardinality content in a single node to enhance clarity of the XML document.

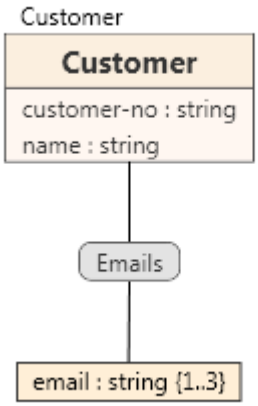
Definition 3.4.8 (Content Container)**.** *A* content container $CC \in \mathcal{CC}$ *is a* subordinate *and* superordinate *node defined as:*

$$CC = (n, Parent, Sub)$$

where:

- $n \in \mathcal{L}$ *is the* name *of CC and models the name of the XML element*
- $Parent \in \mathcal{C}_{sup}$ *is the* parent *of C*
- *Sub is an ordered sequence of components of CC; Sub can be empty*

In XSem-H diagrams content container is depicted as a grey box with round corners and with the value of $CC.n$ inside the box. The components in *Sub* are arranged as children of the content container node in the XSem tree. See Figure 3.10.



(a) Example of content container.

```
<Customer customer-no="###"
          name="###">
  <Emails>
    <email>###</email>
    <email>###</email>
  </Emails>
</Customer>
```

(b) Corresponding XML fragment

Figure 3.10: Usage of Content Container and a Corresponding XML Fragment

Content Choice A content choice is the first construct that models possibilities and options. It is used when the set of XML documents conforming to the modeled XML format is irregular. The content choice is a construct composed of options. The corresponding fragment of each conforming XML document conforms to one of these options.

Definition 3.4.9 (Content Choice)**.** *A* content choice $CCh \in \mathcal{CH}$ *construct is a* superordinate *and* subordinate *node*

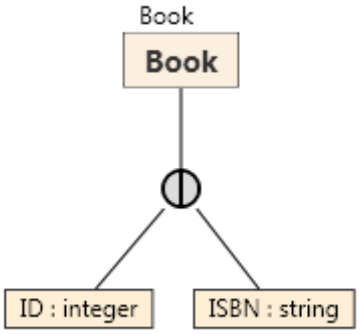
$$CCh = (Parent, Sub)$$

where:

- $Parent \in \mathcal{C}_{sup}$ *is the* parent *of* CCh
- *Sub is an ordered sequence of components of* CCh*; Sub can theoretically be empty, but in that case it is redundant*

In XSem-H diagrams, content choice is depicted as a grey circle with vertical line going through the center. The components in *Sub* are arranged as children of the content choice node in the XSem-H tree. See Figure 3.11. In an XML document the node itself does not have a matching construct, but exactly one of the subtrees under the choice node does. In the example the XML node `Book` can contain either the subelement `ID` or the subelement `ISBN` (we suppose a book

can be identified either by its internal identifier in the system or by its assigned ISBN).



(a) Example of content choice.

```
<Book>
  <ID>###</ID>
</Book>

or

<Book>
  <ISBN>###</ISBN>
</Book>
```

(b) Corresponding XML fragment

Figure 3.11: Usage of Content Choice and a Corresponding XML Fragment

Class Union A class union is a construct related to content choice, because it is also used to model irregularities in the set of XML documents conforming to the modeled format by allowing more types of content.

The difference is that a class union is created by merging two or more associations with common parent. The associations leading to respective classes are removed and replaced by a new association leading from the common parent to the class union. Figure 3.12 shows a diagram before and after introducing new class union node. Two associations `Author-Article` and `Author-Book` are replaced by an association `Author-union(Article, Book)`, classes `Article` and `Book` became components of the new union node.

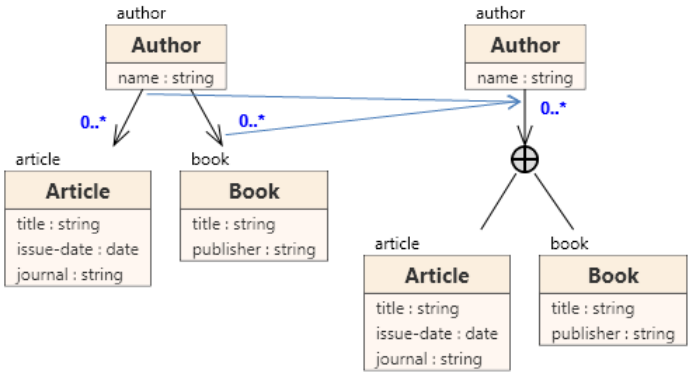


Figure 3.12: PSM Metamodel

Definition 3.4.10 (Class Union)**.** *A* class union *construct is a node*

$$CU = (A, Comp)$$

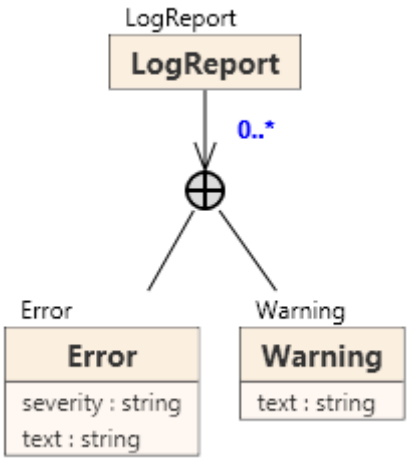
where:

- $A \in \mathcal{A}_{psm}$ *is a PSM association leading to CU*
- $Comp \subset \mathcal{C}_{psm}$ *is an ordered sequence of PSM classes called* components *of the node*

Class union models a selection among a set of options how a particular part of XML document can be structured. Each component (a PSM class) models one of the options.

An association leading to a class union can have a non-default cardinality ($(l, u) \neq (1, 1)$) – this means that the selection can be undertaken multiple times (when $u > 1$) or the whole subtree is optional (when $l = 0$).

In XSem-H diagrams, a class union is depicted as a grey circle with a horizontal and vertical line going through the center and its components arranged in the subtree of the node. Figure 3.13 shows an example of a class union. In this example the modeled node `LogReport` can contain unbounded amount of either `Error` or `Warning` nodes.



(a) Example of class union.

```
<LogReport>
  <Error severity="#" text="#" />
  <Warning text="#" />
  <Error severity="#" text="#" />
</LogReport>

or

<LogReport>
  <Error severity="#" text="#" />
  <Error severity="#" text="#" />
</LogReport>

or

<LogReport />
```

(b) Corresponding XML fragment

Figure 3.13: Usage of Class Union and a Corresponding XML Fragment

Structural Representative In system modeling it is a common requirement to provide constructs for reuse of parts already designed in the model. At the platform-specific layer of XSem this can be achieved by the *structural representative* construct.

Structural representative C is a PSM class that, besides its own attributes and components, references attributes defined in another PSM class in the model. This class is called *represented class* and must be derived from the same PIM class as C (this relation is expressed in the condition $C.Represented \in \{P : \mathcal{C}_{psm} | P.R = C.R\}$ in the definition of PSM class). When $C.Represented$ is empty, C is an ordinary PSM class.

Structural representatives can be useful when:

- a part of the modeled structure is used at several places in the diagram (when the XSem-H diagram is translated to XML Schema language, this situation can be expressed by defining a XML Schema *complex type* and using it for several nodes).
- a part of modeled structure is recursive (since the XSem-H model is a tree, it does not allow cycles of PSM associations).

In XSem-H diagrams, PSM classes that are structural representatives have

blue background instead of orange and the name of represented PSM class ($C.Represented.n$) is shown above the name of the class.

Figure 3.14 provides an example of both of the types of usage of structural representative. For this model, we will show not only an example of an XML document conforming to this model, but also a possible translation of this model to the XML Schema language. Both can be found in Appendix B.

Figure 3.14 shows a whole XSem-H diagram modeling a book catalog. Each book has an author and a title. This structure we need in two places in the diagram – in the listings of books inside a category and for the element best-seller for each category (containing the best selling book in the category). To avoid redefining the structure several times, we use a PSM class `BookType` and reference it from two other PSM classes (with element labels `best-seller` and `book`).

The book catalogue is a hierarchical structure starting with `catalog` element, serving as the "root" category, which has subcategories in `category` element and again each category can have subcategories. Each category also has a textual description and a list of books in the category. To model this hierarchical structure, we again utilize structural representative construct.

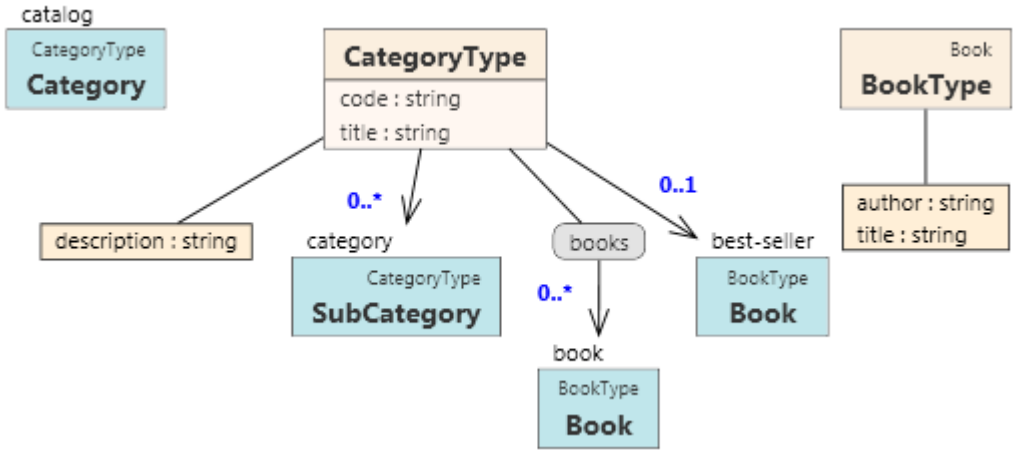


Figure 3.14: Usage of Structural Representative Construct

XSem Tree To conclude this chapter we will formally define the XSem tree.

Definition 3.4.11 (XSem Tree)**.** *An* XSem tree $\mathcal{T}_{PSM}$ *is a rooted, ordered, directed tree*

$$\mathcal{T}_{PSM} = (r, \mathcal{N}, \mathcal{E})$$

where

- $r \in \mathcal{N}$ *is the root node*
- $\mathcal{N}$ *is the set of nodes*
- $\mathcal{E}$ *is the set of edges*

$$\mathcal{N} \subseteq \mathcal{C}_{psm} \cup \mathcal{AC} \cup \mathcal{CC} \cup \mathcal{CH} \cup \mathcal{CU}$$

$$\begin{aligned}\mathcal{E} = \big(& \{(p,c) : p \in \mathcal{C}_{sup}, c \in \mathcal{N}_{child} \mid \\ & \qquad \exists\ A \in \mathcal{A}_{psm} : p = A.Parent \wedge c = p.Child \} \cup \\ & \{(p,c) : p \in \mathcal{C}_{sup}, c \in \mathcal{C}_{sub} \mid c.Parent = p \wedge c \in p.Sub \backslash \mathcal{A}_{psm} \} \cup \\ & \{(u,c) : u \in \mathcal{CU}, c \in \mathcal{N}_{child} \mid c \in u.Comp \}\big) \cap \{(p,c) \mid p,c \in \mathcal{N} \}\end{aligned}$$

The ordering of children of each node is defined by the ordering of collections $p.Sub$ *where* $p \in \mathcal{C}_{sup}$ *and* $u.Comp$ *where* $u \in \mathcal{CU}$ *which are both defined as ordered sequences.*

Nodes in the XSem tree correspond to PSM classes, attribute containers, content containers, content choices and class unions. Edges correspond to PSM associations and also pairs of superordinate nodes and their components and pairs of class unions and their components (these edges are depicted as simple lines, whereas PSM associations are depicted as arrows).

An XSem-H diagram is comprised of one or more XSem trees.

Definition 3.4.12 (XSem-H Diagram)**.** *An* XSem-H diagram $\mathcal{D}$ *is defined as*

$$\mathcal{D} = (roots, \mathcal{N}, \mathcal{E})$$

where $roots \subset \mathcal{C}_{psm} \cup \mathcal{CC}$ *are root nodes of the XSem trees in the diagram.*

Each node $N \in \mathcal{N}$ *in* $\mathcal{D}$ *determines a subtree which will be denoted* $\widehat{N}$*.*

Each significant node in *roots* (content container or PSM class with element label) models one allowed root node of a conforming XML document. PSM classes without element labels can not model root XML nodes, but can be referenced from other XSem subtrees via *structural representative* construct.

Structural representatives can only reference classes from the same diagram, i.e. the following condition must hold:

$$\forall C \in \mathcal{D}.\mathcal{N} \cap \mathcal{C}_{psm} : C.Represented \neq null \rightarrow C.Represented \in \mathcal{D}.\mathcal{N}$$

Chapter 4

XSem Evolution

4.1 Overview

By schema evolution we mean conducting certain operations upon the existing schema until reaching the desired final state – new version of the schema.

Every XSem-H diagram can be translated into a regular tree grammar $G_{\mathcal{D}}$ [23]. Grammar $G_{\mathcal{D}}$ can be obtained for instance by translating $\mathcal{D}$ to XSD [24] and the XSD into regular tree grammar [23].

XSem-H diagram $\mathcal{D}$ is a model for a set of XML documents that are generated by grammar $G_{\mathcal{D}}$.

Definition 4.1.1 (Set of Conforming Documents, Validity)**.** *For a diagram $\mathcal{D}$, the set of conforming documents $\mathcal{S}(\mathcal{D})$ equals to the language $\mathcal{L}(G_{\mathcal{D}})$ generated by grammar $G_{\mathcal{D}}$. We will say that XML document D is* valid *against $\mathcal{D}$ if $D \in \mathcal{S}(\mathcal{D}) = \mathcal{L}(G_{\mathcal{D}})$.*

In the following text we will use apostrophes to mark the evolved model, e.g. $\mathcal{D}'$ for the new version of XSem diagram $\mathcal{D}$.

The degree of difference between the two versions can vary greatly – the newer version can only fix some names, change the arrangement of elements or tweak the data types to be more accurate. On the other hand, the new version of the system can bring new attributes and concepts at the platform-independent level and these will probably emerge also in the updated versions of platform-specific diagrams.

Two special situations can be observed (we will define both at the level of a single PSM diagram):

Definition 4.1.2 (Backward Compatibility)**.** *Let $\mathcal{D}$ be an XSem diagram and $\mathcal{D}'$ new version of $\mathcal{D}$. $\mathcal{D}'$ is called* backwards-compatible *when all documents valid against $\mathcal{D}$ are also valid against $\mathcal{D}'$, i.e. $\mathcal{S}(\mathcal{D}) \subseteq \mathcal{S}(\mathcal{D}')$.*

Definition 4.1.3 (Forward Compatibility)**.** *Let $\mathcal{D}$ be an XSem diagram and $\mathcal{D}'$ new version of $\mathcal{D}$. $\mathcal{D}$ is called* forward-compatible *when all documents valid against $\mathcal{D}'$ are also valid against $\mathcal{D}$, i.e. $\mathcal{S}(\mathcal{D}') \subseteq \mathcal{S}(\mathcal{D})$.*

However, sets $\mathcal{S}(\mathcal{D})$ and $\mathcal{S}(\mathcal{D}')$ are in general incomparable.

Testing validity of a document D against $\mathcal{D}$ can be carried out directly, but it is more convenient to translate $\mathcal{D}$ into one of the XML schema languages and use this translation to verify validity[1].

A translation algorithm that produces schemas in XML Schema language was proposed in [24] and implemented in XCase [7]. Modifications of this algorithm and also algorithms for other XML schema languages are subject of contemporary studies.

Translation of XSem diagrams to an XML schema does not serve only to verify validity of documents but the resulting schemas can also be used in different parts of the system (e.g. to provide descriptions for web service interfaces [9] or to design optimal storage for XML documents in a relational database [2]).

The fundamental problem of the system and in our case schema evolution is integration with those components accustomed to the old version of the system. The first part of the problem is deciding, whether the changes in the schema may cause problems with these components.

Definition 4.1.4 (Invalidated Set of Conforming Documents)**.** *We say that the set of conforming documents $\mathcal{S}(\mathcal{D})$ of diagram $\mathcal{D}$ was* invalidated in the new version *(or just* invalidated*) if:*

$$\exists D \in \mathcal{S}(\mathcal{D}) : D \notin \mathcal{S}(\mathcal{D}')$$

If no such D exists, then $\mathcal{D}'$ is backwards-compatible.

Definition 4.1.5 (Revalidation)**.** *For an invalidated set of conforming documents the process of adjusting the invalid documents to the new version is called* revalidation*:*

$$\forall D \in \mathcal{S}(\mathcal{D}), D \notin \mathcal{S}(\mathcal{D}') : revalidate(\mathcal{S}(\mathcal{D})) \in \mathcal{S}(\mathcal{D}')$$

[1]Each XML schema language has its own specifics, some aspects of the XSem model can not be utterly translated to the means of the schema language (e.g. XSem allows to model an element having either attribute a or attribute b but not both, which can not be expressed in XML Schema language).

4.2 Approaches to Change Detection

For the goal of determining whether $\mathcal{S}(\mathcal{D})$ was invalidated, the system must recognize and analyze the differences between $\mathcal{D}$ and $\mathcal{D}'$. There are two possible ways to recognize changes

a) recording the changes as they are conducted during the design process

b) comparing the two versions of the diagram

Recording Changes An evolution system that uses the first technique usually provides some kind of command that initiates the recording and after issuing this command all operations carried out by a user over the schema are recorded. The user starts with schema S and initiates recording. All conducted operations leading to S' are recorded until the user is satisfied with the new schema. When the desired schema S' is reached, user finishes recording and the system has all the information about the changes made – the sequence of performed operation.

When the recording is finished, the system can optimize the sequence for example by eliminating operations that cancel each other or by replacing groups of operations by other groups that lead to the same result but in a more straight way. These optimizing rules must be defined in the system. This approach is used in CoDEX [17].

The main problem of this approach is the insufficient versatility. The following issues may arise:

- Once the evolution process is started, the old version can not be easily changed.
- A user may want to interrupt his/hers work at some point and continue in another session. The sequence of recorded changes would have to be stored and recording resumed later.
- When the user wants to retrieve the sequence for reverse process, he will have to either start with the new version and record the operations needed to go back to the old version again, or the system will have to be able to create inverse sequence for each sequence of operations.
- When the evolved schema comes from an outer source, the sequence of operation changes can not be retrieved directly; the user must start with his/hers old version of the schema and manually adjust it to match the new schema.

Schema comparison An alternative approach, used in this work, is to base the change detection on comparison of the two versions. The user can work with both schemas independently until he/she is satisfied with them. The change detection algorithm then takes the two schemas as input and compares them. The result of the comparison is a list of differences between the schemas.

This approach has the following advantages:

- No need to look for redundancies; the set of changes is always minimal.
- Both old version and new version can be edited without limitations.
- The process of evolution can be arbitrarily stopped and resumed.
- The reverse operation can be easily handled by the same algorithm, only with the two schemas on the input swapped.
- A schema from an outer source can be imported into the system[2] and serve as an input to the change detection algorithm.

The approach assumes the two version of the schema to be linked. The algorithm requires constructs from one version of the schema be linked to constructs to the other version. These links are created and kept as user edits the schema, but do not exist when one or both versions of the schema are imported. In such case, the links must be created either by the user or heuristically by another algorithm comparing the two versions.

4.3 Version Links

Detecting changes in an XML schema or a model of an XML schema is not always straightforward; some differences in between the old and new version can be interpreted in more than one way. Consider the following excerpt from two versions of an XSem diagram:

Figure 4.1: Example for Ambiguity

[2]Experimental implementation of import from XSD to PSM diagram is available in XCase.

There are at least two possible interpretations:

- attribute `ID` was removed and new attribute `security-number` was added
- attribute `ID` was renamed to `security-number`

When deciding which interpretation is the correct one, links to PIM diagram can be taken into account. If $[ID].r = p1$ and $[security - number].r = p2$ and both $p1$ and $p2 \in \mathcal{A}tt_{pim}$ such that $p2$ is the new version of $p1$, we could assume, that the second interpretation is correct and attribute was only renamed. But still, it is only a heuristic.

If we do not want to settle for heuristics, it is the user who must make the final decision. XCase evolution framework (XSem-Evo) allows to define version links to interconnect constructs in the XSem model which are different versions of the same construct.

Definition 4.3.1 (ver, version link, getInVer, version projection)**.** *Let $\mathcal{V}$ be the set of versions for the model $\mathcal{M}$.*

Function

$$ver : \mathcal{M} \rightarrow \mathcal{V}$$

returns to which version each construct belongs. Instead of writing $ver(e)$ we will use the notion $e.version$.

Notion $\mathcal{S}[v]$, where $\mathcal{S} \subseteq \mathcal{M}$ is an arbitrary set of model constructs and $v \in \mathcal{V}$, is called version projection *and returns elements of $\mathcal{S}$ for whose the condition $e.version = v$ holds.*

En equivalence relation of version links

$$\mathcal{VL} \subset \mathcal{M} \times \mathcal{M}$$

contains pairs of constructs that are different versions of the same construct.

Function getInVer returns a construct in a desired version (or null if the element does not exist in the desired version)

$$\begin{aligned} getInVer &: (\mathcal{M} \times \mathcal{V}) \rightarrow \mathcal{M} \cup null \\ getInVer(e, v) &= \begin{cases} e' & \text{if } \exists\, e' \in \mathcal{M} : (e, e') \in \mathcal{VL} \ \wedge \ e'.version = v \\ null & \text{otherwise} \end{cases} \end{aligned}$$

If $(e, e') \in \mathcal{VL}$, both e and e' must both be constructs of the same kind (e.g. both classes, attributes, PSM associations...).

In the following text we will assume $|\mathcal{V}| = 2$, unless explicitly stated otherwise (i.e. we expect there are two versions in the system - old version ($v \in \mathcal{V}$) and new version ($v' \in \mathcal{V}$)).

Values of *ver* function form the input of the change detection algorithm (and so does the relation $\mathcal{VL}$). In XCase, values of *ver* are assigned automatically by the system during the process of editing the new version. Pairs for $\mathcal{VL}$ are added during the *branch* operation (a new version v' is created from version v), which adds new version to $\mathcal{V}$ and creates a copy of each construct from the source version.

Branch is an operation upon model with the following results ($v \in \mathcal{VL}$ is the source version):

- new version v' is added to $\mathcal{V}$
- $\forall e \in \mathcal{M}[v]$:
  - a duplicate construct e' is created in the model
  - $e'.version$ is set to v'
  - (e, e') is added to $\mathcal{VL}$
- $\forall e' \in \mathcal{M}[v']$ all references to other construct in the model are replaced by references to new versions of each construct (e.g. if e' is a *content container*, each $c \in e'.Sub$ is replaced by $c' = getInVer(c, v')$)

When the new version of the schema is imported into the system, i.e. the new version was not initiated by performing *branch* operation upon the model, the relation $\mathcal{VL}$ is empty and version links must be added before change detection. This can be either done manually (ie. we let the user map new versions of the elements to their old versions) or with the help of heuristics – the system can try to match the pairs of elements based on their types, names and placement in the diagram.

In conclusion, an XSem model with multiple versions contains two kinds of links – inter-layer links connecting PIM and PSM constructs (in the form of properties of PSM constructs that reference PIM constructs) and version links connecting different versions of the same construct. Figure 4.2 illustrates this.

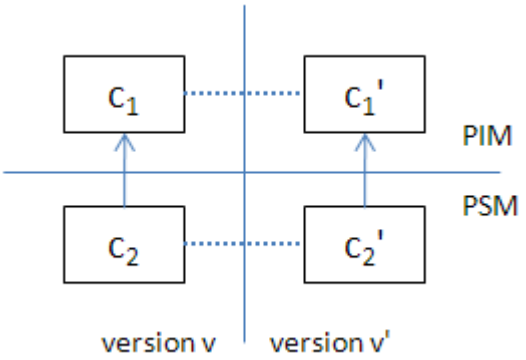


Figure 4.2: Model Links

In this figure there is one PIM construct (c_1) and one PSM construct (c_2, derived from c_1) in two versions, PIM-PSM links are drawn as vertical arrows, version links as dotted lines, $\mathcal{VL}$ would for this model be $\{(c_1, c_1'), (c_2, c_2')\}$.

The version link between c_2 and c_2' may seem to be redundant, but without it some situations could be misinterpreted. See Figure 4.3 for an example. In this example, the PIM remained unchanged but in the PSM model, PSM class `Employee` was removed and replaced by a new class (also named `Employee` but with different element label and cardinality). This is indicated by the absence of version link between the pair of Employee classes. The old diagram modeled `Department` element with sub-element `head-of-department`. The new version models department with a list of all employees. If we did not rely on the version links at the platform-specific layer but only on the version links at the platform-independent layer, it would not be possible to distinguish this situation (i.e. a class was removed, another class was added) from the situation, where the class is an adjusted version of the class from the previous version.

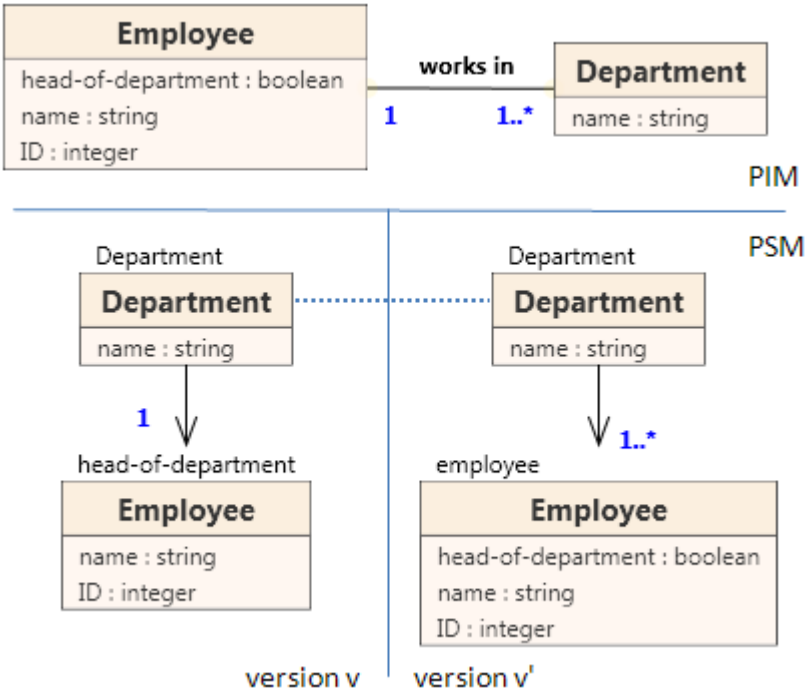


Figure 4.3: Version Links

4.4 Approaches to Revalidation

When the new version $\mathcal{D}'$ of the diagram $\mathcal{D}$ is created, in general case it will invalidate the set $\mathcal{S}(\mathcal{D}')$. After examining the set of changes, the evolution algorithm can

- decide, whether $\mathcal{S}(\mathcal{D})$ was invalidated
- alternatively provide a revalidation instruction

Definition 4.4.1 (revalidation script)**.** *Revalidation script is a script or sequence of commands that, when executed upon any document $D \in \mathcal{S}(\mathcal{D})$, produces a document $D' \in \mathcal{S}(\mathcal{D}')$.*

As regards the implementation language for the revalidation script, there are several options:

- Instruction for DOM API [34]; the document is loaded into memory and the instruction executed by a tool implementing the DOM API.

- XQuery Update Facility script [38]; An XQuery Update Facility processor performs the update commands from revalidation script on $\mathcal{D}$ leaving $\mathcal{D}'$ as a result.

- XSL transformation [16], An XSLT processor performs the transformation (the revalidation script) with $\mathcal{D}$ as input and $\mathcal{D}'$ as output.

- SQL and SQL/XML script [14] – when the documents in $\mathcal{S}(\mathcal{D})$ are stored in a relational database and obtained by SQL/XML, the script could take form of SQL data definition language commands and SQL data manipulation language commands to reflect the changes in the database and then the SQL/XML scripts that were used to retrieve the documents in the old version will be replaced by SQL/XML scripts ready for the new version.

XSem-Evo uses XSLT as revalidation script implementation language due to the wide support for XSLT among the tools working with XML data and, especially, the database systems supporting XML Schema evolution.

Chapter 5

Changes Between Versions

In this chapter we will specify changes that can occur between two versions of a PSM diagram, examine their impact on validity of documents and propose a correction in the cases where validity is violated.

A *Change detection* algorithm is responsible for comparing the two versions of an XSem-H diagram and finding the set of changes between them.

Changes are defined as predicates, each having certain amount of parameters, the first parameter always corresponds to one of the scopes enumerated above. For an n-tuple of constructs satisfying a change predicate c we will use a notation $\tilde{c}$ and call it an *instance* of a change predicate c. The list of instances of predicates is the output of the change detection algorithm.

5.1 Changes Categorization

We will divide the set of changes into four groups according to the character of a change (classification is similar to [21]):

- addition – new construct was added
- removal – a construct was removed from the model
- migratory – a construct (and its subtree in the XSem-H tree) was moved to another part of the XSem-H tree
- sedentary – an existing construct was adjusted in place, but not moved

Change Scope For each change there is also defined a type of construct, where it could be detected (e.g. PSM class construct for *class name change*), called scope. Here is the list of all scopes, each scope with a list of XSem constructs:

- diagram: XSem diagram
- subordinate: association, attribute container, content choice, content container
- superordinate: class, content choice, content container
- construct with attributes: class, attribute container
- association target: class, class union
- class: class
- association: association
- class union: class union
- attribute: attribute

Changes defined for scope s will be denoted $changes_s$. Some constructs act as several types of scope (e.g. class construct must be examined by change detection algorithm as superordinate, construct with attributes, association target and class scope).

In the following text we assume $v, v' \in \mathcal{V}$ to be the two compared versions and $\mathcal{D}'$ the new version of the diagram $\mathcal{D}$. Constructs marked with apostrophe belong to $\mathcal{M}[v']$, constructs without the apostrophe mark belong to $\mathcal{M}[v]$.

Each of the following sections is devoted to the changes in respective scopes listed above. Description of each change predicate contains explanations the parameters of the predicate parameters, the definition of the predicate definition and a list of actions required to revalidate the document when the change occurs.

5.1.1 Diagram Changes

Diagram Root Added A new root is added in the diagram using an addition change with the following parameters:

- $C' \in \mathcal{D}'.roots$: root added in version v'
- $i' \in \mathbb{N}_0$: index of C' in $\mathcal{D}'.roots$ collection

Definition

$$\big(C' \in \mathcal{D}'.roots \wedge getInVersion(C', v) = null \wedge \\ i' = \mathcal{D}'.roots.index(C')\big) \leftrightarrow diagramRootAdded(C', i')$$

Revalidation Adding a new diagram root does not violate validity, see Section 5.3.

Diagram Root Removed A root was removed from the diagram using a removal change with the following parameters:

- $C \in \mathcal{D}.roots$: root removed in version v'

Definition

$$C \in \mathcal{D}.roots \wedge getInVersion(C, v') = null \leftrightarrow diagramRootRemoved(C')$$

Revalidation To revalidate a diagram with removed root, a new root must be selected. It must be one of the significant nodes from $\mathcal{D}'.roots$. Before the root class was removed, the user may have moved some constructs to a different XSem-H tree and, thus, some parts of the existing content may be preserved (removing a diagram root does not necessary result in discarding the whole document).

Diagram Root Index Change The index of a root node in the roots sequence changed using a migratory change with the following parameters:

- $C' \in \mathcal{D}'.roots$: root whose index changed
- $i' \in \mathbb{N}_0$: new index of C' in the collection $\mathcal{D}'.roots$

Definition

$$\big(C' \in \mathcal{D}'.roots \wedge getInVersion(C', v) \neq null \wedge i' = D'.roots.index(C') \wedge \\ i' \neq \mathcal{D}.roots.index(getInVersion(C', v))\big) \\ \leftrightarrow diagramRootIndexChange(C', i')$$

Revalidation Order of the nodes in *roots* sequence does not violate validity of the document since only one of the root nodes can represent the root XML element of the XML document, see Section 5.3.

5.1.2 Subordinate Node Changes

Subordinate abstract construct has a meta-property *Parent* pointing to its parent node in the XSem-H tree. This parent can be a subject of change when the construct is moved either to another node or to another position within the same superordinate node. In the following text, let S' be the examined subordinate construct (and S its previous version, if it exists).

Subordinate Component Index Change The order of S within collection $S.Parent.Sub$ was changed in version v' using a migratory change with the following parameters:

- $S' \in \mathcal{C}_{sub}$: examined subordinate node
- $i' \in \mathbb{N}_0$: new index of S in the collection $S'.Parent.Sub$

Definition

$$\begin{gathered}\big(getInVersion(S', v) \neq null \wedge \\ getInVersion(S'.Parent, v) = getInVersion(S.Parent, v') \wedge \\ i' = S'.Parent.Sub.index(S') \wedge i' \neq S.Parent.Sub.index(S)\big) \\ \leftrightarrow subordinateComponentIndexChange(S', i')\end{gathered}$$

Revalidation The situation again differs greatly from the case where $S'.Parent$ is a content choice. If so, no revalidation is needed, because the order of components under content choice has no impact on validity of documents.

When $S'.Parent$ is a class or a content container, it is crucial, whether S' or its subtree models any XML elements (S' is a *significant node* or subtree of S' contains at least one *significant node*). As long as S' and its subtree only models attributes, validity is not impaired, because XML data model specifies that the order of attributes of an XML node is unimportant and no application should rely on attributes being defined in some particular order.

Thus only when $S'.Parent$ is not a content choice and S' is or its subtree contains any *significant node*s, revalidation is needed. In that case, all content modeled by S' and its subtree must be moved to its correct position i among the content modeled by other constructs (siblings) in $S'.Parent.Sub$.

Subordinate Component Moved Subordinate component S was moved from its parent $P = S.Parent$ (and P' is its new version, if it exists) to another superordinate node $Q' = S'.Parent$ using a migratory change with the following parameters:

- $S' \in \mathcal{C}_{sub}$: examined subordinate node
- $Q' \in \mathcal{C}_{sup}$: new parent of S (superordinate node) in version v', $S' \in Q'.Sub$
- $i' \in \mathbb{N}_0$: new index of S' in the collection of $Q'.Sub$

Definition

$$\begin{gathered}\big(getInVersion(S', v) \neq null \wedge \\ getInVersion(S'.Parent, v) \neq getInVersion(S.Parent, v') \wedge \\ Q' = S'.Parent \wedge i' = Q'.Sub.index(S')\big) \\ \leftrightarrow subordinateComponentMoved(S', Q', i')\end{gathered}$$

Revalidation All content modeled by S' must be removed from P' (regardless S' being a content container, content choice or class). The way of incorporating this content into the content of Q' depends on the type of construct Q.

If Q' is a class or a content container, content is copied.

If Q' is a content choice and if there exists some content modeled by another construct in $Q'.Sub$, there are two semantically correct solutions. Since Q' is a content choice, there can be only one such construct, let it be O'. See Figure 5.1 for an example.

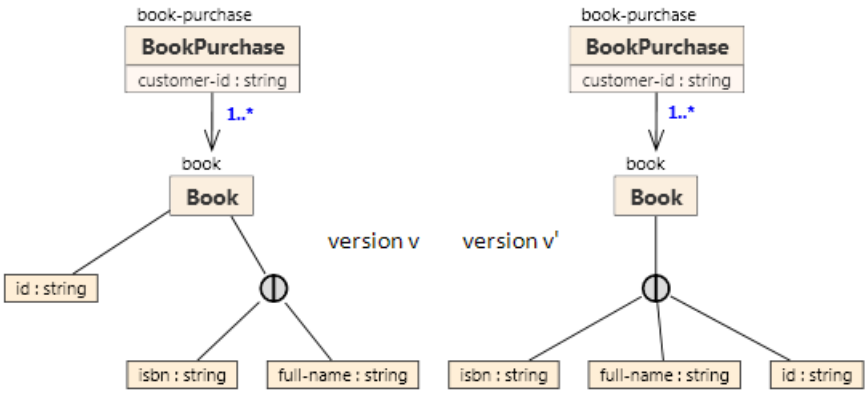


Figure 5.1: Moving a Construct to Content Choice

Figure 5.1 depicts a simple PSM diagram D_c of a book purchase XML format. Subelement `book` is required to have a subelement `id` and one of the subelements `isbn` or `full-name`. In the new version, `id` is no longer required and is now one of the options under content choice (so exactly one of the three subelements can appear).

Each XML document $D \in \mathcal{S}(D_c)$ will have the `id` attribute and one of the attributes in the content choice. Since the user decided to move the `id` attribute under the content choice, it is now equally correct to either preserve the `id` attribute or one of the other two.

The revalidation algorithm must either choose which solution it will prefer:

- Always preserve the content modeled by the construct newly moved to choice (S') – in the example attribute `id`. But if there are more components moved to the same choice, this rule is still not decisive enough.
- Conversely, prefer the construct that already was in the choice (O').
- Set the priorities in the order of the components of the content choice – in the example this order would be `isbn`, `full-name`, `id`)

or leave the decision up to the user.

5.1.3 Superordinate Node Changes

A *Superordinate* node is an abstract construct defined to involve a collection of *subordinate* components *Sub*. Changes detectable in the scope of a superordinate node concern changes in the collection as well. In the following text, let P' be the examined superordinate node (and P its previous version, if it exists).

Component Added New component was added into collection $P'.Sub$. using an addition change with the following parameters:

- $P' \in \mathcal{C}_{sup}$: examined superordinate node
- $C' \in P'.Sub$: component added in the new version to $P'.Sub$
- $i' \in \mathbb{N}_0$: new index of C' in the collection of components of P'

Definition

$$\big(C' \in P'.Sub \wedge getInVersion(C', v) = null \wedge i' = P'.Sub.index(C')\big) \\ \leftrightarrow subordinateComponentAdded(P', C', i')$$

Revalidation In general, when a component was added, the content modeled by this component has to be created and inserted into each document $D \in \mathcal{S}(\mathcal{D})$. The following alternatives can emerge:

- P' is a content choice – in this case, adding a component can be completely ignored, because the new component for content choice provides the user with more options, but does not force him/her to use them all (see Section 5.3).
- P' is a PSM class or a content container – in this case, the change can not be ignored in general (but still can be in some special cases, e.g. when C' is an association with $l = 0$), C' including its subtree must be examined and the respective content must be created in the closest *significant node* to C'.

Component Removed Component C was removed from the collection $P'.Sub$ using a removal change with the following parameters:

- $P' \in \mathcal{C}_{sup}$: examined superordinate node
- $C \in P.Sub$: component removed from $P'.Sub$

Definition

$$\big(C \in P.Sub : getInVersion(C, v') = null\big) \\ \leftrightarrow subordinateComponentRemoved(P', C)$$

Revalidation Removal of a construct from the model must be always solved by removal of the content modeled by the removed construct from the documents $\in \mathcal{S}(\mathcal{D})$.

However, the content modeled by the whole subtree of C can not be instantly removed from the document, because some other changes may move parts of this content to other parts of the document.

5.1.4 Changes of Association Target

Set $\mathcal{N}_{child} = \mathcal{C}_{psm} \cup \mathcal{CU}$ contains nodes that can be targets of PSM associations – classes and class unions. Similarly to subordinate constructs, these can be moved either from/to a class union or (theoretically) from one association to another. Also the order of components under the class union can change.

Change of Class Index The order of class C within $C.U.Comp$ changed in version v' using a migratory change with the following parameters:

- $C' \in \mathcal{C}_{psm}$: examined class
- $U' \in \mathcal{CU}$: class union to which C' belongs
- $i' \in \mathbb{N}_0$: new index of C in the union components collection $U'.Comp$

Definition

$$\big(getInVer(C', v) \neq null \wedge$$
$$U' = C'.U \neq null \wedge getInVer(U', v) \neq null \wedge$$
$$getInVer(U', v) = getInVer(C', v).U$$
$$i' = U'.Comp.index(C') \wedge i' \neq C.U.Comp.index(C)\big)$$
$$\leftrightarrow classIndexChange(C', U', i')$$

Revalidation Similarly to reordering components of a content choice, reordering components of a class union requires no revalidation, see Section 5.3.

Class Moved to a Class Union Class C that is a target node of an association A is moved to a class union node U' using a migratory change with the following parameters:

- $C' \in \mathcal{C}_{psm}$: examined class
- $U' \in \mathcal{CU}$: class union to which C' is moved
- $i' \in \mathbb{N}_0$: new index of C' in the components collection $U'.Comp$ of the union

Definition

$$\big(C = getInVer(C', v) \neq null \wedge U' = C'.U \neq null \wedge$$
$$i' = U'.Comp.index(C') \wedge$$
$$(C.A \neq null \vee C.A = C.U = null \vee C.U \neq getInVer(U', v) \neq null)\big)$$
$$\leftrightarrow classMovedToClassUnion(C', U', i')$$

Revalidation Moving a class to a class union requires a similar action as moving a subordinate construct to a content choice (see paragraph Subordinate Component Moved on page 62). The content of C and its subtree must be removed from components of $A.Parent$ (if A is not empty) and either added to the contents of the closest *significant node* to U' or discarded completely (the possibilities are identical to those for moving subordinate component to a content choice).

Class Moved out of a Class Union In version v, class C was among the components of a class union U. In version v' class C' is changed into a target node of an association A' or a root class using a migratory change with the following parameters:

- $C' \in \mathcal{C}_{psm}$: examined class
- $A' \in \mathcal{A}_{psm}$: association leading to C' in version v' of the diagram (empty when C' is a root class)

Definition

$$\begin{aligned}\big(C = getInVer(C', v) \neq null \wedge C.U \neq null \wedge A' = C'.A \wedge \\ (C'.A \neq null \vee C'.A = C'.U = null)\big) \\ \leftrightarrow classMovedOutOfClassUnion(C', A')\end{aligned}$$

Revalidation When a class C is moved out of a class union U in the new version, there are two possible situations for each document $D \in \mathcal{S}(\mathcal{D})$ (providing that $\mid U.Comp \mid \geq 2$ which is presumable):

- the corresponding part of D is valid against the subtree modeled by C (the component C of U is used by the document)
- the corresponding part of D is valid against another subtree modeled by $O \in U.Comp : O \neq C$ (the component C is not used by the document).

In the first case, the content modeled by $\widehat{C}$ is removed from the closest *significant node* of U and moved to the closest *significant node* of A'. If $getInVer(U, v') \neq null$ (U was not completely removed in the new version), content modeled by $\widehat{C}$ must be replaced by content modeled by a selected $\widehat{O'}, O' \in U'.Comp$. This content can be retrieved either from some other part of document D (when O' was moved into the class union in the new version):

$$\exists A_2 s.t.\ classMovedToClassUnion(O', U', A_2, i') \in C_{\mathcal{D},\mathcal{D}',v,v'} : A_2 \neq null$$
$$\rightarrow use\ O'$$

If no suitable O' can be found, new content must be generated to revalidate the document.

In the second case, there is no content modeled by $\widehat{C}$ in D, so the content for $\widehat{C'}$ must be generated.

Cardinality There are two associations participating in this change – association A' which leads to C' and the association that leads to U, let us denote it A_u. Both of these associations can have non-default ($\neq (1,1)$) cardinality. The cardinality values for both associations must be respected in the revalidation process, which means that

- some data may have to be discarded (when $A_u.u > A'.u$);
- some data may have to be generated (when $A_u.l < A'.l$).

The situation depends on the contents of each individual $D \in \mathcal{S}(\mathcal{D})$ and the revalidation script must be ready for all alternatives.

Class Moved To Another Association Class C was a target node of an association $B \neq getInVer(A', v)$ in version v, but in version v' it was changed into a target node of an association A' using a migratory change with the following parameters:

- $C' \in \mathcal{C}_{psm}$: examined class
- $A' \in \mathcal{A}_{psm}$: association leading to C' in version v' of the diagram

Definition

$$\big(A' = C'.A \neq null \wedge C = getInVer(C', v) \neq null \wedge C.A \neq null \wedge$$
$$A' \neq C.A\big) \leftrightarrow classMovedToAnotherAssociation(C', A')$$

Revalidation Content modeled by C' must be removed from its old location (closest significant node to $B.Parent$) and moved to the contents modeled by $A'.Parent$.

Class Moved To Roots Class C' is a root class, but in version v it was a target node of some association A or among components of some class union U. using a migratory change with the following parameters:

- $C' \in \mathcal{C}_{psm}$: examined class
- $i' \in \mathbb{N}_0$: new index of C' in the collection $\mathcal{D}'.roots$

Definition

$$\big(C' \in \mathcal{D}'.roots \wedge C = getInVer(C', v) \wedge i' = \mathcal{D}'.roots.index(C') \wedge (C.A \neq null \vee C.U \neq null)\big) \leftrightarrow classMovedToRoots(C', i')$$

Revalidation If $C'.label \neq null$, moving an existing class C to roots collection of a diagram triggers making the instance of class C the root element of the XML document. If $C'.label = null$, instance of class C is removed from its old location. If $C.U \neq null$, it must be taken into account that there may not be an existing instance of C in the document.

5.1.5 Changes of Classes

In the following text, let $C' \in \mathcal{C}_{psm}$ be a PSM class (and C its previous version, if it exists).

Class Given an Element Label C' was assigned with an element label, in the previous version, C did not have an element label using a sedentary change with the following parameters:

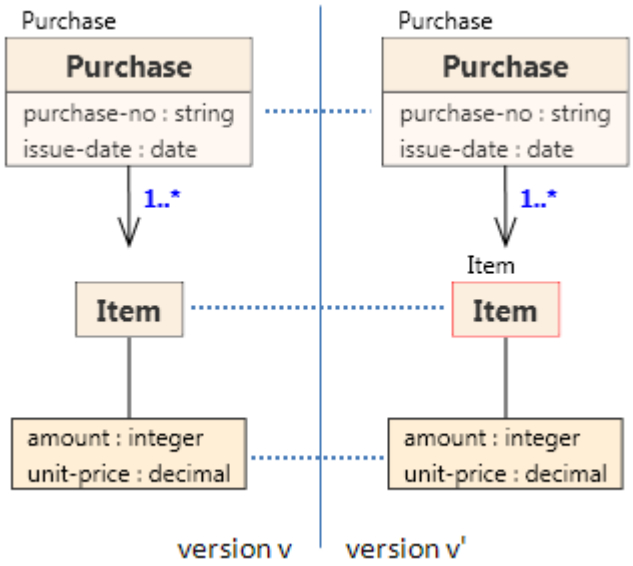
- $C' \in \mathcal{C}_{psm}$: class assigned with an element label in the new version
- $label' \in \mathcal{L}$: new element label of class C'

Definition

$$\big(getInVer(C', v) \neq null \wedge getInVer(C', v).label = null \wedge label' = C'.label \neq null\big) \leftrightarrow classGivenElementLabel(C', label')$$

Revalidation Attributes and elements modeled by a class without an element label are propagated to the closest *significant node*. Let be P' be the closest *significant node* of $C.Parent'$ and Q the closest *significant node* of C (not necessarily $getInVer(P', v) = Q$). When C is assigned with an element label, new XML element N must be created as a subelement of P' (its name will be the value of $C'.label$), attributes modeled by $\widehat{C'}$ will model XML attributes of element N and elements modeled by $\widehat{C'}$ will become subelements of N.

Cardinality Association A' leading to C' or association leading to the class union U' (when $C' \in U'.Comp$) can have non-default cardinality. In this case, each instance of $\widehat{C'}$ in XML document D must be treated separately, i.e. new subelement N_i is created for the instance i. See Figure 5.2 – A' is between `Purchase` and `Item`, C' is `Item`. Each instance of $\widehat{\texttt{Item}}$ – pair of `amount` and `unit-price` elements – is wrapped in the new element `Item`.



(a) Adding element label

```
<Purchase purchase-no="1"
    issue-date="1993-03-17">
    <amount>2</amount>
    <unit-price>
        9.27
    </unit-price>
    <amount>4</amount>
    <unit-price>
        11.34
    </unit-price>
</Purchase>
```

(b) Document valid for v

```
<Purchase purchase-no="1"
    issue-date="1993-03-17">
  <Item>
    <amount>2</amount>
    <unit-price>9.27</unit-price>
  </Item>
  <Item>
    <amount>4</amount>
    <unit-price>11.34</unit-price>
  </Item>
</Purchase>
```

(c) Document revalidated for v'

Figure 5.2: Example – Adding of an Element Label

Element Label of a Class Removed In version v C had an element label, in version v' the element label was removed from class C' using a sedentary change with the following parameters:

- $C' \in \mathcal{C}_{psm}$, class from which element label was removed

Definition

$$\big(getInVer(C', v) \neq null \wedge getInVer(C', v).label \neq null \wedge C'.label = null\big) \\ \leftrightarrow classElementLabelRemoved(C')$$

Revalidation The needed revalidation is an exact opposite of revalidation of *classGivenElementLabel* change. The wrapping XML element corresponding to C (its name being the value of *C.label*) is removed, its subelements inlined and its attributes moved to the closest *significant node* of C'. The example in Figure 5.2 can be used for illustration, only with the versions v and v' swapped.

Element Label of Class Changed The element label of class C was changed using a sedentary change with the following parameters:

- $C' \in \mathcal{C}_{psm}$, class with changed element label
- $label' \in \mathcal{L}$: new element label of class C'

Definition

$$\big(getInVer(C', v) \neq null \wedge \\ getInVer(C', v).label \neq null \wedge C'.label \neq null \\ label' = C'.label \wedge label' \neq getInVer(C', v).label\big) \\ \leftrightarrow classElementLabelChanged(C', label')$$

Revalidation For each instance of content modeled by C, the XML element (name being the value of *C.label*) must be renamed to *label′* (attributes and subelements are left intact).

5.1.6 Content Container Changes

The *content container* construct has a meta-property *name* (n) modeling the name of the XML element wrapping contents of the content container. Changes in the other meta-properties – *Parent* and *Sub* – were covered earlier in this chapter in subsections Subordinate Node Changes (p. 63) and Superordinate Node Changes (p. 61).

In the following text, let CC' be the examined subordinate construct (and CC its previous version, if it exists).

Content Container Name Change Content container CC was renamed using a sedentary change with the following parameters:

- $CC' \in \mathcal{CC}$: examined content container
- $n' \in \mathcal{L}$: new content container name

Definition

$$\begin{gathered}\big(getInVersion(CC', v) \neq null \wedge \\ n' = CC'.n \wedge n' \neq getInVersion(CC', v).n\big) \\ \leftrightarrow contentContainerNameChange(CC', n')\end{gathered}$$

Revalidation Revalidation for this change is simple - each XML element modeled by CC named $CC.n$ must be renamed to $CC'.n$ in the XML document.

5.1.7 Attribute Changes

In the following text, let a' be the examined PSM attribute (and a its previous version, if it exists).

Attribute Alias or Name Change A PSM attribute has two different meta-properties that influence the name of the modeled XML attribute – n and *alias*. Name is acquired from PIM ($\forall(a_1, a_2) \in \mathcal{A}tt_{psm} \times \mathcal{A}tt_{psm} : a_1.R = a_2.R \leftrightarrow a_1.n = a_2.n = a_1.R.n$) and unchangeable; that is why the optional *alias* property is provided and *alias* (if specified) overrides n.

We will define auxiliary function *aliasOrName*:

Definition 5.1.1 (aliasOrName)**.**

$$aliasOrName : \mathcal{A}tt_{psm} \leftrightarrow \mathcal{L}$$

$$aliasOrName(a) = \begin{cases} alias & \text{if } alias \neq null \\ n & \text{otherwise} \end{cases}$$

and use the notion $a.aliasOrName$ *for* $aliasOrName(a)$*.*

The change is a sedentary change with the following parameters:

- $a' \in \mathcal{A}tt_{psm}$: examined attribute
- $n' \in \mathcal{L}$: new attribute name
- $alias' \in \mathcal{L}$: new attribute alias

Definition

$$\begin{gathered}\big(getInVersion(a', v) \neq null \wedge \\ (alias' = a'.alias \wedge alias' \neq getInVersion(a', v).alias \vee \\ n' = a'.n \wedge n' \neq getInVersion(a', v).n)\big) \\ \leftrightarrow attributeAliasOrNameChange(a', n', alias')\end{gathered}$$

Revalidation Revalidation for this change is simple – each XML attribute modeled by a (named $a.aliasOrName$) must be renamed to $a'.aliasOrName$ in the XML document. In the case of $a.alias$ being defined and $a.alias = a'.alias \neq null$, revalidation in the XML documents is not needed (only n was changed, but the property n is overridden by *alias*).

Attribute Index Change In the XSem-H model, collections of PSM attributes are ordered sequences and the index of the attribute a ($a.i$) can vary from version to version.

The change is a migratory change with the following parameters:

- $a' \in \mathcal{A}tt_{psm}$: examined attribute
- $i' \in \mathbb{N}_0$: new index of the attribute

Definition

$$\begin{gathered}\big(getInVersion(a', v) \neq null \wedge \\ getInVersion(a', v).Container = a'.Container \\ i' = a'.i \wedge i' \neq getInVersion(a', v).i\big) \\ \leftrightarrow attributeIndexChange(a', i')\end{gathered}$$

Revalidation Revalidation depends on the type of construct $a'.Container$. If it is a *PSM class*, then a' models XML attribute and since the order of attributes in an XML element is insignificant and applications must not rely on the order of attributes, no revalidation is needed.

On the other hand, if $a'.Container$ is an *attribute container*, the order of PSM attributes determines the order of XML subelements, which is significant. Reordering PSM attributes in an attribute container thus requires reordering of the subelements modeled by the attributes with respect to the new order.

Attribute Default Value Change Default value $a'.d$ is changed in version v' using a sedentary change with the following parameters:

- $a' \in \mathcal{A}tt_{psm}$: examined attribute
- $d' \in dom_{a'.T}$: new default value of the attribute

Definition

$$\begin{gathered}\big(getInVersion(a', v) \neq null \wedge \\ d' = a'.d \wedge d' \neq getInVersion(a', v).d\big) \\ \leftrightarrow attributeDefaultValueChange(a', d)\end{gathered}$$

Revalidation Changing a default value requires no revalidation. The new default value can be used when generating new content during revalidation.

Moving Attribute Attribute a' was moved to an attribute container A' or class C' (in the previous version it was in class C_o or another attribute container A_o). This change does not cover moving an attribute to another position within the same class or attribute container (see paragraph Attribute Index Change (p. 73)).

The change is a migratory change with the following parameters:

- $a' \in \mathcal{A}tt_{psm}$: examined attribute
- $C' \in \mathcal{C}_{psm} \cup \mathcal{AC}$: attribute container into which a' was added
- $i' \in \mathbb{N}_0$: index of a' in $A'.\mathcal{A}tt_{psm}$

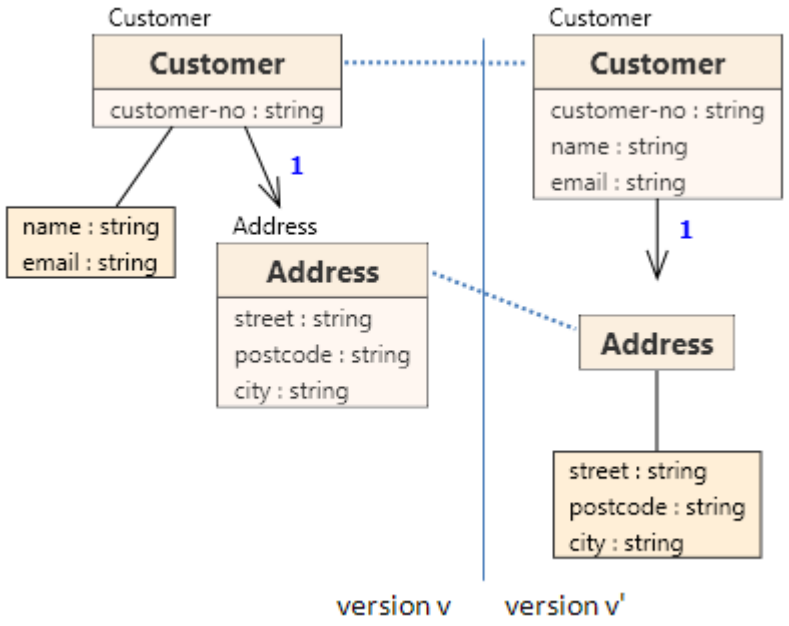
Definition

$$\begin{gathered}\big(getInVersion(a', v) \neq null \wedge i' = a'.Container.\mathcal{A}tt.index(a') \wedge \\ C' = a'.Container \wedge C' \neq getInVersion(a', v).Container\big) \\ \leftrightarrow attributeMoved(a', C', i')\end{gathered}$$

Revalidation Moving a PSM attribute a requires:

- deleting the instance of a from its previous location (it can be either an XML attribute – if $a.Container \in \mathcal{C}_{psm}$, or an XML element with simple content – if $a.Container \in \mathcal{AC}$).
- create a new XML node of the respected type in the new location (either a new XML attribute – if $a'.Container \in \mathcal{C}_{psm}$ or an XML element – if $a'.Container \in \mathcal{AC}$).

Figure 5.3 shows an example of moving attributes from a class to an attribute container (`street`, `postcode`, `city`) and from an attribute container to a class (`name`, `email`). In version v' element label of class `Address` is removed, thus the subelements are propagated to the element `Customer`.



(a) Moving attributes

```
<Customer customer-no="12">
  <name>John Smith</name>
  <email>john@smith.org</email>
  <Address street="Sunset Blvd."
    postcode="12345" city="LA"/>
</Customer>
```

(b) Document valid for v

```
<Customer customer-no="12"
  name="John Smith"
  email="john@smith.cz">
  <street>Sunset Blvd.</street>
  <postcode>12345</postcode>
  <city>LA</city>
</Customer>
```

(c) Document Revalidated for v'

Figure 5.3: Example – Moving Attributes Between Classes and Attribute Containers

Attribute Type Change The new version can specify a different type T' for an attribute instead of the former type T using a sedentary change with the following parameters:

- $a' \in \mathcal{A}tt_{psm}$: examined attribute
- $T' \in \mathcal{T}$: new type of the attribute

Definition

$$\begin{gathered}\big(getInVersion(a', v) \neq null \wedge \\ T' = a'.Type \wedge T' \neq getInVersion(a', v).Type\big) \\ \leftrightarrow attributeTypeChanged(a', a'.Type)\end{gathered}$$

Revalidation Revalidation of documents may be skipped in the case where $dom_T \subseteq dom_{T'}$. This condition is guaranteed if T is a type derived from T' using restriction. The condition means that the requirements for the documents were relaxed and more general set of values is allowed.

Such a situation may occur, but the reverse direction in the development is more frequent. Instead of a general set of values, the requirements are made more strict and only a specific subset of values is allowed. E.g. instead of an arbitrary string for `email` attribute in the old version, only strings valid against a regular expression describing all the possible email addresses are allowed in the evolved version.

In such case, either

$$dom_T \supseteq dom_{T'}$$

or the two sets are incomparable. Let us denote $[a][\mathcal{S}(\mathcal{D})]$ the set of all values of attribute a in all documents in $\mathcal{S}(\mathcal{D})$, we can extend the previous approach if:

$$[a][\mathcal{S}(\mathcal{D})] \subseteq dom_{T'}$$

In this case no revalidation is needed again. Verifying this condition can not be possible in every case, however in some situations, it can be done easily. When we return to the email example, the XML schema may define email as an arbitrary string, but the system contains another component that verifies each email more strictly, before it can occur in a document $D \in \mathcal{S}(\mathcal{D})$. The applicability of this approach can be decided by the user.

If revalidation is really necessary, a revalidating function $conv_a$:

$$\begin{aligned} conv_a : dom_T \quad & \leftrightarrow dom_{T'} \\ & \textit{or} \\ conv_a : [a][\mathcal{S}(\mathcal{D})] \quad & \leftrightarrow dom_{T'} \end{aligned}$$

must be provided for the revalidation algorithm. The function $conv_a$ can be either shared for the pairs of types $(\forall (a, a') \in \mathcal{A}tt_{psm} \times \mathcal{A}tt_{psm} \cap \mathcal{VL} : a.Type =$

$T \wedge a'.Type = T' \leftrightarrow conv_a = conv_{T,T'}$, $conv_{T,T'}$ being the shared revalidation function for types T and T') or defined separately for each attribute a.

Attribute Cardinality Change A cardinality interval ($a.l$, $a.u$) is changed to ($a'.l$, $a'.u$). using a sedentary change with the following parameters:

- $a' \in \mathcal{A}tt_{psm}$: examined attribute
- $l' \in \mathbb{N}_0$: new *lower cardinality*
- $u' \in \mathbb{N}_0 \cup \{*\}$: new *upper cardinality*

Definition

$$\begin{gathered}\big(getInVersion(a', v) \neq null \wedge l' = a'.l \wedge u' = a'.u \wedge \\ (getInVersion(a', v).l \neq l' \vee getInVersion(a', v).u \neq u')\big) \\ \leftrightarrow attributeCardinalityChanged(a', l', u')\end{gathered}$$

Revalidation For cardinality changes, there are two revalidation actions, from which none, one or both must be undertaken to revalidate a document (varying from document to document).

- if $a'.l > a.l$, new content may have to be added for some documents
- if $a'.u < a.u$, content may have to be removed from some documents

For each document $D \in \mathcal{S}(\mathcal{D})$, the number of XML nodes (elements or attributes, depending on $a.Container$ being a PSM class or an attribute container) that are instances of a differs (unless $a.l = a.u$), therefore the amount of XML nodes that need to be added/removed differs too.

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

When adding nodes, the values for these nodes must be assigned. In real world applications, raising the lower cardinality from $a.l \geq 1$ to $a'.l > a.l$ is rare, but yet possible. On the other hand, making an optional subelement/attribute mandatory is a routine practice (from $a.l = 0$ to $a'.l > a.l$). That is why approaches to generating values of attributes need to be discussed (for this example, a suitable solution would be using a default value of the attribute – $a.d$).

5.1.8 Changes in Attribute Collections

A collection of PSM attributes is a part of XSem-H constructs PSM class and Attribute container. Attributes and elements can be added/removed to/from the collection. In the following text, let $N' \in \mathcal{C}_{psm} \cup \mathcal{AC}$ be a PSM class or an attribute container (and N its previous version).

Attribute Added A new attribute was added into the collection (attribute did not exist in the previous version) using an addition change with the following parameters:

- $N' \in \mathcal{C}_{psm} \cup \mathcal{AC}$: examined construct
- $a' \in N'.\mathcal{A}tt$: new attribute added in version v'
- $i \in \mathbb{N}_0$: index of attribute a' in the owner collection

Definition

$$\begin{aligned}\big(a' \in N'.\mathcal{A}tt_{psm} \wedge getInVersion(a', v) = null \wedge \\ i' = a'.Container.\mathcal{A}tt.index(a')\big) \\ \leftrightarrow attributeAdded(N', a', i')\end{aligned}$$

Revalidation If attribute a' is not added as optional ($a'.l = 0$), new content must be added into the document – either an XML element with simple content (if $a.Container \in \mathcal{AC}$) or an XML attribute (if $a.Container \in \mathcal{C}_{psm}$). More about generating content can be found in Section 5.4.

Attribute Removed An attribute was removed from the model completely using a removal change with the following parameters:

- $N' \in \mathcal{C}_{psm} \cup \mathcal{AC}$: examined construct
- $a \in N.\mathcal{A}tt$: removed attribute

Definition

$$\begin{aligned}a \in N.\mathcal{A}tt_{psm} \wedge getInVersion(a', v') = null \\ \leftrightarrow attributeRemoved(N', a)\end{aligned}$$

Revalidation All instances of attribute a must be removed from the document (XML attributes or XML elements with simple content).

5.1.9 Class Union Changes

Classes can be added or removed from a class union. In the following text, let $CU' \in \mathcal{CU}$ be a class union (and CU its previous version).

Class Added to a Class Union A new component was added into the collection (the component did not exist in the previous version) using an addition change with the following parameters:

- $CU' \in \mathcal{CU}$: examined class union
- $C' \in \mathcal{C}_{psm}$: new class added to the union in version v'
- $i' \in \mathbb{N}_0$: index of class C' in the union

Definition

$$\big(C' \in CU'.Comp \wedge getInVersion(C', v) = null \wedge$$
$$i' = C'.U.Comp.index(C')\big) \leftrightarrow classAddedToUnion(CU', C', i')$$

Revalidation Adding a class to a class union does not require revalidation of documents, but only gives more choices when creating new documents.

Class Removed from Class Union A class in a class union was removed from the model using a removal change with the following parameters:

- $CU' \in \mathcal{CU}$: examined class union
- $C \in \mathcal{C}_{psm}$: removed PSM class

Definition

$$\big(C \in CU.Comp \wedge getInVersion(C, v') = null\big)$$
$$\leftrightarrow classRemovedFromUnion(N', C)$$

Revalidation If $CU'.A.l = 0$ (where $CU.A$ is the association leading to class union), the content modeled by the union is optional and its simple removing is sufficient to revalidate the document.

If $CU'.A.l > 0$ and there is an instance of C in the XML document, it must be removed and replaced by an instance of another component $O' \in CU'.Comp$. In some cases, the content modeled by O' can be retrieved from some other part of the document. It is possible when O – the old version of O' – was moved to the union CU' in this version ($O \notin CU.Comp$) and there exists an instance of CU. In other cases, content must be generated.

Class Union Moved A class union node is moved from one place in the diagram to another using a migratory change with the following parameters:

- $CU' \in \mathcal{CU}$: examined class union
- $A' \in \mathcal{A}_{psm}$: association leading to CU' in version v'

Definition

$$\begin{gathered}\big(getInVersion(CU', v) \neq null \ \wedge A' = CU'.A \\ getInVer(A', v) \neq getInVer(CU', v).A\big) \\ \leftrightarrow classUnionMoved(CU', A')\end{gathered}$$

Revalidation Since union nodes themselves do not have corresponding content in the XML documents (only their components have), revalidation is discussed with the changes in the union components collection.

5.1.10 Association Changes

In the following text, let A' be a PSM association (and A its previous version).

Association Cardinality Change A cardinality interval ($A.l$, $A.u$) is changed to ($A'.l$, $A'.u$). using a sedentary change with the following parameters:

- $A' \in \mathcal{A}_{psm}$: examined PSM association
- $l' \in \mathbb{N}_0$: new *lower cardinality*
- $u' \in \mathbb{N}_0 \cup \{*\}$: new *upper cardinality*

Definition

$$(getInVersion(A', v) \neq null \wedge l' = A'.l \wedge u' = A'.u$$
$$(getInVersion(A', v).l \neq l' \vee getInVersion(A', v).u \neq u'))$$
$$\leftrightarrow associationCardinalityChanged(A', l', u');$$

Revalidation Similarly as with *attribute cardinality change*, there exist two revalidation actions, from which none, one or both must be undertaken to revalidate a document (varying from document to document).

- if $A'.l > A.l$, new content may have to be added for some documents
- if $A'.u < A.u$, content may have to be removed from some documents

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

For each document $D \in \mathcal{S}(\mathcal{D})$, the number of instances of $A.Child$ differs (unless $A.l = A.u$), therefore the amount of instances of A that need to be added/removed differs too.

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

When adding, the content for the new instances must be generated (this involves generating a whole XML subtree). More about generating content can be found in Section 5.4.

A Node Added Under Association A new class or class union node is added into the diagram using an addition change with the following parameters:

- $A' \in \mathcal{A}_{psm}$: examined PSM association
- $C' \in \mathcal{CU} \cup \mathcal{C}_{psm}$: added node

Definition

$$getInVer(A', v) \neq null \ \wedge \ C' = A'.Child \ \wedge \ getInVer(C', v) = null$$
$$\leftrightarrow associationChildAdded(A', C')$$

Revalidation Adding a node under a PSM association requires creating new content (the amount of instances required is determined by $A'.l$).

A Node Removed Under Association A Class or class union node is removed from the diagram using a removal change with the following parameters:

- $A' \in \mathcal{A}_{psm}$: examined PSM association
- $C = A.Child \in \mathcal{CU} \cup \mathcal{C}_{psm}$: removed node

Definition

$$\begin{gathered}\big(getInVer(A', v) \neq null \ \wedge \\ C = A.Child \ \wedge \ getInVer(C, v') = null\big) \\ \leftrightarrow associationChildRemoved\,(A', C)\end{gathered}$$

Revalidation Removal of a node under a PSM association requires complete removal of all the content modeled by the node.

5.2 Changes Summary

Definition 5.2.1 (Set of Changes)**.** *The set of all changes between the two versions v and v' of a diagram ($\mathcal{D}$ and $\mathcal{D}'$) will be denoted $C_{\mathcal{D},\mathcal{D}',v,v'}$.*

Table 5.1 lists all changes with their respective types grouped by scope.

Scope	Name	Type
Diagram	diagramRootAdded	Addition
	diagramRootRemoved	Removal
	diagramRootIndexChange	Migratory
Subordinate	subordinateComponentIndexChange	Migratory
	subordinateComponentMoved	Migratory
Superordinate	subordinateComponentAdded	Addition
	subordinateComponentRemoved	Removal
Association target	classIndexChange	Migratory
	classMovedToClassUnion	Migratory
	classMovedOutOfClassUnion	Migratory
	classMovedToRoots	Migratory
Class	classGivenElementLabel	Sedentary
	classElementLabelRemoved	Sedentary
	classElementLabelChanged	Sedentary
	classAnyAttributeAllowed	Sedentary
	classAnyAttributeForbidden	Sedentary
Content container	contentContainerNameChange	Sedentary
Attribute	attributeAliasOrNameChange	Sedentary
	attributeIndexChange	Migratory
	attributeDefaultValueChange	Sedentary
	attributeMoved	Migratory
	attributeTypeChange	Sedentary
	attributeCardinalityChange	Sedentary
Construct with attributes	attributeAdded	Addition
	attributeRemoved	Removal
Class Union	classAddedToUnion	Addition
	classRemovedFromUnion	Removal
Association	associationCardinalityChange	Sedentary
	associationChildAdded	Addition
	associationChildRemoved	Removal

Table 5.1: Changes Summary

5.3 Changes Not Affecting Validity

One set of changes consists of changes that do not affect validity of the existing documents. It may seem futile to be concerned with these changes and we can simply ignore them, but being able to detect any changes in the XSem tree can be useful for purposes other than translating to XML schema.

Even though a change $\tilde{c} \in C_{\mathcal{D},\mathcal{D}',v,v'}$ does not violate validity of $\mathcal{S}(\mathcal{D})$, it will

affect the translation algorithm, which implies that the XML schema translated from the diagram $\mathcal{D}$ will differ from the schema translated from $\mathcal{D}'$.

E.g. reordering components of content choice has no effect on validity, because in a valid XML document only one of the possible choices will be selected. Still, the translated schemes will differ in the order of declarations in the content of appropriate `<xs:choice>` (assuming the XML Schema language is utilized for translation). Addition of a new component into content choice also has no effect on validity, but the change is in this case very significant. Assuming that the XML documents are shredded into a set of tables of a relational database, this set must be altered to handle storing those documents that will use the newly added component in content choice in the future.

Familiarity with changes not affecting validity can significantly simplify the process of revalidation. Of course, if we are sure that all changes made in the evolved schema belong to the set of changes not affecting validity, it is correct to skip the revalidation of $\mathcal{S}(\mathcal{D})$, because validity against the new schema is guaranteed.

Changes not affecting validity can be divided into these groups:

- changes applied only during creation of new elements/attributes in the XML document
- changes giving more options and possibilities than the old schema
- changes broadening cardinality interval
- changes adding optional content
- changes caused by differences between XSem model and XML model.

For the set of changes not affecting validity of the documents in $\mathcal{S}(\mathcal{D})$ we will use the notion $C^{NI}_{\mathcal{D},\mathcal{D}',v,v'}$. Each type of changes not affecting validity will be defined as a set or subset of instances of one or more change predicate. For example, the following statement:

$$\forall\ attributeDefaultValueChange\ \tilde{c} : \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'}$$

says that each instance $\tilde{c}$ of *attributeDefaultValueChange* is a change not affecting validity. The following statement:

$$\forall\ subordinateComponentAdded\ \tilde{c}, \tilde{c}.P' \in \mathcal{CH} : \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'}$$

says that only those instances of *subordinateComponentAdded* change modifying a content choice are changes not affecting validity.

5.3.1 Changes Applied During Creation

If a change creates structures that are taken into account only for newly created elements/attributes, validity of the set of existing documents is not violated.

Assigning/Modifying Default Value

$$\forall\ attributeDefaultValueChange\ \tilde{c} : \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'} \tag{5.1}$$

XSem data model allows to assign default values to attributes. Attributes in XSem model either attributes of elements (when declared in PSM classes) or elements with simple content (when defined inside an attribute container).

When a default value is assigned to an attribute that did not have a default value in the original schema, all affected elements/attributes from the set of existing documents must already have their values specified. The change does not violate validity of the set of existing documents.

If an attribute had a default value in the original schema and it was modified in the evolved schema, the value of the affected elements/attributes in the set of existing documents that do not have their values specified changes. This change is not expressed in the XML documents themselves, but in the XML schema so no revalidation is needed.

5.3.2 Changes Giving More Choices

Adding a new member to a set from which members are selected in the XML document gives users creating and modifying documents more choices in the future, but does not violate validity of the documents already created. Several changes can be put into this category

Adding more classes to class union/adding more components to content choice

$$\forall\ classAddedToUnion\ \tilde{c} : \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'}$$
$$\forall\ subordinateComponentAdded\ \tilde{c}, \tilde{c}.P' \in \mathcal{CH} : \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'} \tag{5.2}$$

A new class in *Class Union* allows for more possible types of content in an XML element. All previous possibilities are preserved and thus validity of the set of existing documents is not violated. Analogously for adding components to *Content Choice*.

Allowing more top level elements

$$\forall\ diagramRootAdded\ \tilde{c}, \tilde{c}.C' \in \mathcal{CC} \cup \{C \in \mathcal{C}_{psm} : C.label = null\} :$$
$$\tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'} \tag{5.3}$$

XSem subtrees with roots being content containers or PSM classes having an element label are added in the PSM diagram.

Adding an XSem-H subtree with content container or PSM Class having an element label as a root allows another root element in the valid documents. Newly created documents can be created following the new XSem subtree, but the set of existing documents remains valid.

Until some component of the new subtree is referenced from one of the subtrees existing in the previous versions (e.g. via means of *structural representative* construct), the presence of a new subtree has no effect on existing documents (new documents can of course be created as instances of the new subtree).

Adding XSem subtrees with a root class without an *element label*

$$\forall\ diagramRootAdded\ \tilde{c}, \tilde{c}.C' \in \mathcal{CC} \cup \{C \in \mathcal{C}_{psm} : C.label \neq null\} :$$
$$\tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'} \tag{5.4}$$

Adding an XSem-H subtree with a root being a PSM class without *label* defines new type of content that can not appear in any document in the set of existing documents.

Until some component of the new subtree is referenced from one of the subtrees existing in the previous versions (e.g. via means of *structural representative* construct), the presence of a new subtree has no effect on existing documents.

Changing the type of an attribute to a more general type

$$\forall\ attributeTypeChange\ \tilde{c}, dom_{getInVer(\tilde{c}.a',v).Type} \subseteq dom_{\tilde{c}.T'} : \\ \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'} \tag{5.5}$$

If type T of attribute a is changed to T' in $\mathcal{D}'$ and $dom_T \subseteq dom_{T'}$, then all values used in $\mathcal{S}(\mathcal{D})$ valid against $\mathcal{D}$ remain valid against $\mathcal{D}'$ and $\mathcal{D}'$ is backwards compatible.

Depending on tghe particular storage of XML documents, some additional changes can be also declared non-invalidating. If documents in the set of existing documents are stored as text (e.g as XML files or in a *CLOB* column in a relational database), values of all attributes are stored as text. In that case any type can be changed to `string` (or `normalizedString` and possibly user-defined types derived from `string`) without invalidating $\mathcal{S}(\mathcal{D})$.

In other scenarios (e.g. when values of an attribute typed `integer` are stored in a *NUMBER* column in a relational database), changing a type of an attribute to `string` may invalidate the set of existing documents.

5.3.3 Changes Broadening Cardinality Interval

$$\forall\ associationCardinalityChange\ \tilde{c}, \\ getInVer(\tilde{c}.A', v).l \geq \tilde{c}.l' \ \wedge\ getInVer(\tilde{c}.A', v).u \leq \tilde{c}.u' : \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'}$$

$$\forall\ attributeCardinalityChange\ \tilde{c}, \\ getInVer(\tilde{c}.a', v).l \geq \tilde{c}.l' \ \wedge\ getInVer(\tilde{c}.a', v).u \leq \tilde{c}.u' : \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'} \tag{5.6}$$

XSem defines the cardinality of PSM associations as a pair (l, u), where $l \in \mathbb{N}_0$ and $u \in \mathbb{N} \cup \{*\}$ and inequality $l \leq u$ always holds ($*$ being greater than any l). Cardinality of attributes is defined in a similar manner.

The cardinality interval (l, u) can be changed in the evolved schema to (l', u'). If conditions:

$$\begin{aligned} l' &\leq l \\ u' &\geq u \end{aligned} \tag{5.7}$$

hold, it is guaranteed that the XML fragments valid against the original schema remain valid for the evolved schema.

If one of the conditions in 5.7 is violated, it is impossible to decide whether some of the documents from $\mathcal{S}(\mathcal{D})$ will become invalid. Even narrowing the cardinality interval can still leave all the documents in $\mathcal{S}(\mathcal{D}')$ valid (e.g. when $l = 5$ and $l' = 4$ for an attribute a in an attribute container c, but the element <a> does not occur for five times in any document $D \in \mathcal{S}(\mathcal{D})$).

5.3.4 Changes Adding Optional Content

It is possible to view the changes that add optional content as a special case of changes broadening the cardinality interval (using the same terms as above, it means that $l = 0$, $u = 0$, $l' = 0$ and $u' > 0$) but this perspective can be rather unnatural to the user, so we will discuss them separately.

When XML schema evolution takes a form of adding new content and backward compatibility is desired, adding new content as optional is a recommended approach (for more recommendation on writing backward and forward compatible schemas and queries see [22]).

The whole set of existing documents will lack the newly added content, but will remain valid due to the content being declared as optional.

Adding optional attribute, association

$$\begin{aligned} &\forall\ attributeAdded\ \tilde{c}, \tilde{c}.a'.l = 0 : \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'} \\ &\forall\ subordinateComponentAdded\ \tilde{c}, \tilde{c}.C' \in \mathcal{A}_{psm}\ \wedge\ \tilde{c}.C'.l = 0 : \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'} \end{aligned} \tag{5.8}$$

Each attribute a' and association A' in XSem model has its cardinality (l', u'). If a new attribute/association is added having $l' = 0$, it will model an optional content. In case of attributes, the form being either attribute of an element or an element with simple content depending on whether the attribute is inside a PSM class or an attribute container respectively. In case of associations the modeled content is either a PSM subtree when $A.Child$ is a *significant node* or a part of the content of the closest *significant node* of $A.Child$.

5.3.5 Utilizing *Any Attribute* Flag

$$\begin{aligned}
&\forall\ classAnyAttributeAdded\ \tilde{c} : \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'} \\
&\forall\ attributeAdded\ \tilde{c}, \tilde{c}.Container' \in \mathcal{C}_{psm}\ \wedge \\
&\qquad getInVer(\tilde{c}.Container', v).a = true : \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'}
\end{aligned} \tag{5.9}$$

Adding *Any Attribute* flag to a PSM Class or adding an attribute where *Any Attribute* is declared belong to this category.

Allowing *Any Attribute* allows the user to add an arbitrary attribute to the significant ancestor of the class. Validity of the set of existing documents is not impaired.

Allowing *Any Attribute* for a *PSM Class* can be an efficient way to give the users a certain level of extensibility when working with the schema and achieving the *forward-compatibility*. When *Any Attribute* was defined in the original schema, attributes can be added arbitrarily in the evolved schema and the set of existing documents will remain valid.

5.3.6 Changes Caused by Differences between XSem Model and XML Model

Some properties of the constructs in the XSem model do not directly influence the content of the set $\mathcal{S}(\mathcal{D})$.

Reordering Attributes in PSM Classes

$$\forall\ attributeIndexChange\ \tilde{c}, \tilde{c}.a'.Container' \in \mathcal{C}_{psm} : \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'} \tag{5.10}$$

Since attributes in PSM classes model XML attributes and the order of XML attributes in an XML element is insignificant, this change does not violate validity.

Reordering Classes in Class Unions/Components in Content Choices/Diagram roots

$$\forall\ diagramRootIndexChange\ \tilde{c} : \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'}$$
$$\forall\ classIndexChange\ \tilde{c} : \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'}$$
$$\forall\ subordinateComponentIndexChange\ \tilde{c}, \tilde{c}.S' \in \mathcal{CH} : \tilde{c} \in C^{NI}_{\mathcal{D},\mathcal{D}',v,v'} \quad (5.11)$$

Reordering collections of content choice/class union nodes has no effect on the validity of XML data.

5.4 Generating Content

Certain modifications in the diagram may require new content to be added into some (or all) documents in $\mathcal{S}(\mathcal{D})$ to revalidate the documents:

- new non-optional construct is added in the diagram (*subordinateComponentAdded*, *attributeAdded*)
- cardinality interval was extended from (l, u) to (l', u') where $l < l'$ (*attributeCardinalityChange*, *associationCardinalityChange*)
- a construct was moved or deleted from content choice or non-optional class union and its instance in the XML document must be replaced by instance of one of the other components in the content choice/classes in the class union (*subordinateComponentMoved*, *classMovedOutOfClassUnion*).

This section will propose solutions to this issue.

5.4.1 User-Provided Content

Leaving the issue up to the user to solve it can be very convenient and in some cases the only correct solution. The revalidation script for evolution from version v to v' can be generated by the system with some "blanks" left for the user to fill or take a form of a parameterized script with the values of parameters to be filled when executing the script.

Figure 5.4 contains a simple example and Figure 5.5 shows a revalidation script that references additional document (in Figure 5.6) to provide the necessary data. New attribute `email` was added to element customer and a document `CustomerData.xml` contains list of emails of all customers.

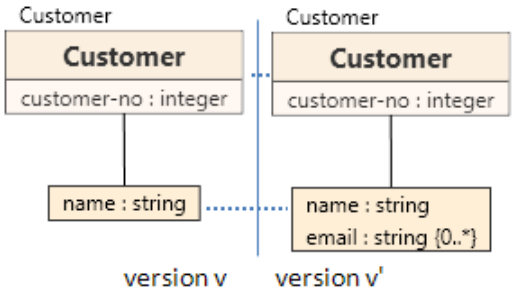


Figure 5.4

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="2.0"
  xmlns:xsl="..">
  <xsl:output method="xml" indent="yes" />

  <xsl:variable name="cl"
       select="document('CustomerList.xml')/Customers" />

  <xsl:template match="/Customer">
    <Customer>
      <xsl:copy-of select="@name" />
      <xsl:copy-of select="@customer-no" />
      <xsl:variable name="cn" select="@customer-no" />
      <xsl:copy-of select="$cl/Customer[@customer-no = $cn]/email" />
    </Customer>
  </xsl:template>
</xsl:stylesheet>
```

Figure 5.5: Revalidation stylesheet

```
<?xml version="1.0"?>
<Customers>
  <Customer customer-no='34'>
    <email>john.doe@example.org</email>
    <email>john.doe@company.com</email>
  </Customer>
  ...
  <Customer customer-no='77'>
    <email>martin.smith@domain.org</email>
  </Customer>
</Customers>
```

Figure 5.6: Document Referenced from Revalidation Stylesheet

5.4.2 Default Content

The revalidation script can also create the missing content itself – the structure of the internal nodes is given, the values for leaf nodes and attributes can be supplied with the default values of PSM attributes (if defined) or the default values for the given type of each PSM attribute (an empty string for `xs:string` type etc.). Where choices/unions are present, the first component/class is always selected. Such a content will be called *default instance*.

Here we will describe an algorithm that returns a default instance of an XSem-H subtree with root N. Each time a content is needed to be generated, the default instance can be created for the particular subtree.

First, we will show how to obtain a *default instance grammar* $G^{def}_{\mathcal{D}}$ from an arbitrary XSem-H diagram $\mathcal{D}$. Then, with this grammar, a default instance of any subtree in diagram $\mathcal{D}$ can be generated from a particular nonterminal node.

For grammar $G^{def}_{\mathcal{D}}$ let $\mathcal{N}$ be the set of nonterminals and $\mathcal{T}$ the set of terminals.

$$
\begin{aligned}
\mathcal{T} = &\{l | \exists\ \textit{PSM class}\ C \in \mathcal{D}.\mathcal{N} : C.\textit{label} = l\} \cup \\
&\{n | \exists\ \textit{content container}\ CC \in \mathcal{D}.\mathcal{N} : CC.n = n\} \cup \\
&\{a_n | \exists\ \textit{PSM attribute}\ a \in \bigcup_{H:\mathcal{C}_{psm} \cup\ \mathcal{AC}} H.\mathcal{A}tt_{psm} : a.n = a_n\} \cup \\
&\{a_d | \exists\ \textit{PSM attribute}\ a \in \bigcup_{H:\mathcal{C}_{psm} \cup\ \mathcal{AC}} H.\mathcal{A}tt_{psm} : a.d = a_d\} \cup \\
&\{T_d | \exists\ \textit{type}\ T \in \mathcal{T} : T_d\ \textit{is a default value for}\ T\} \cup \\
&\{<, />, \text{␣}, \text{attribute}=\}
\end{aligned}
$$

The terminals are labels of all classes, names of all content containers and attributes, default values for all attributes and types and the set of strings in the last line.

The set of nonterminals $\mathcal{N}$ will consists of:

- $\widehat{N}, \widehat{N}^{elm}, \widehat{N}^{att}$ where
  $N \in \mathcal{C}_{psm} \cup \mathcal{AC} \cup \mathcal{CC} \cup \mathcal{CH} \cup \mathcal{CU}$
- $\widehat{A}^{elm}, \widehat{A}^{att}$ where $A \in \mathcal{A}_{psm}$
- $\widehat{a}$ *for each PSM attribute* a
- $\widehat{A}^{elm}_{exp}$ for each association $A \in \mathcal{A}_{psm}$ and $\widehat{a}^{elm}_{exp}$ for each PSM attribute a

Productions then follow the structure of XSem-H tree:
For each class C_l with element label, attributes $a_1, ..., a_k$ and components $S_1, ..., S_m$ add rules:

$$\widehat{C_l} \rightarrow < C_l.label\ \widehat{a_1}...\widehat{a_k}\ \widehat{S_1}^{att}...\widehat{S_m}^{att} > \widehat{S_1}^{elm}...\widehat{S_m}^{elm} < /C_l.label >$$
$$\widehat{C_l}^{elm} \rightarrow \widehat{C_l}$$

For each class C_{nl} without element label, attributes $a_1, ..., a_k$ and components $S_1, ..., S_m$ add rules:

$$\widehat{C_{nl}}^{att} \rightarrow \widehat{a_1}...\widehat{a_k}\ \widehat{S_1}^{att}...\widehat{S_m}^{att}$$
$$\widehat{C_{nl}}^{elm} \rightarrow \widehat{S_1}^{elm}...\widehat{S_m}^{elm}$$
$$\widehat{C_{nl}} \rightarrow {C_{nl}}^{elm}\ {C_{nl}}^{att} \qquad (*)$$

For each content container CC with components $S_1, ..., S_m$ add rules:

$$\widehat{CC} \rightarrow < CC.n\ \ \widehat{S_1}^{att}...\widehat{S_m}^{att} > \widehat{S_1}^{elm}...\widehat{S_m}^{elm} < /CC.n >$$
$$\widehat{CC}^{elm} \rightarrow \widehat{CC}$$

For each content choice CCh with components $S_1, ..., S_m$ add rules:

$$\widehat{CCh}^{att} \rightarrow \widehat{S_1}^{att}$$
$$\widehat{CCh}^{elm} \rightarrow \widehat{S_1}^{elm}$$
$$\widehat{CCh} \rightarrow CCh^{elm}\ CCh^{att} \qquad (*)$$

For each class union CU with class $C_1, ..., C_m$ add rules:

$$\widehat{CU}^{att} \rightarrow \widehat{C_1}^{att}$$
$$\widehat{CU}^{elm} \rightarrow \widehat{C_1}^{elm}$$
$$\widehat{CU} \rightarrow CU^{elm}\ CU^{att} \qquad (*)$$

For each attribute container AC with class $a_1, ..., a_k$ add rules:

$$\widehat{AC}^{att} \rightarrow \lambda$$
$$\widehat{AC} \rightarrow \widehat{a_1}...\widehat{a_k}$$
$$\widehat{AC}^{elm} \rightarrow \widehat{AC}$$

For each association A with target N add rules:

$$\widehat{A}^{elm} \rightarrow \widehat{A}^{elm}_{exp}...\widehat{A}^{elm}_{exp}\ (\textit{where } \widehat{A}^{elm}_{exp} \textit{ is repeated } A.l \textit{ times})$$
$$\widehat{A}^{att} \rightarrow \widehat{N}^{att}$$
$$\widehat{A}^{elm}_{exp} \rightarrow \widehat{N}^{elm}$$

For each attribute a inside an attribute container add rules:

$$\widehat{a} \rightarrow \widehat{a}^{elm}_{exp}...\widehat{a}^{elm}_{exp}\ (\textit{where } \widehat{a}^{elm}_{exp} \textit{ is repeated } a.l \textit{ times})$$
$$\textit{if } a.d \neq \textit{null}:$$
$$\widehat{a}^{elm}_{exp} \rightarrow < a.aliasOrName > a.d < /a.aliasOrName >$$
$$\textit{otherwise}:$$
$$\widehat{a}^{elm}_{exp} \rightarrow < a.aliasOrName > T_d < /a.aliasOrName >$$

For each attribute a of a PSM class add rules:

$$
\begin{array}{ll}
\textit{if } a.d \neq \textit{null}: & \\
\widehat{a} \rightarrow \textit{attribute}=\text{`}a.d\text{'} & \textit{otherwise}: \\
\widehat{a} \rightarrow \textit{attribute}=\text{`}T_d\text{'} &
\end{array}
$$

The last two rules generate an XML attribute with value. Note that there are only nonterminals on the right-hand side of the rules.

To generate an instance of an XSem-H subtree with root N, which is a significant node, just start with the nonterminal $\widehat{N}$. For each nonterminal, there is only one rule with the given nonterminal on the left-hand side, thus the grammar $G_{\mathcal{D}}^{def}$is deterministic.

When N is not a significant node, the result of generation will be a set of attribute definitions (as a result of propagation to significant nodes) and an XML fragment (rules marked with $(*)$ are the beginning rules when the starting nonterminal corresponds to nonsignificant node).

5.4.3 Utilization of PIM Links

When a new XSem-H construct is added, it also has links to the new version of the platform-independent model. These links can be used to obtain the correct data when content is generated for the new construct.

One example could be the situation where data is stored in a relational database. The content for XSem-H construct can be then retrieved from the data in the database. This would be possible if there is a model of a relational database linked to the PIM. This approach will be a subject of future work.

5.5 Oracle In-Place XML Schema Evolution

This section shows the relation of XSem changes non-invalidating to In-Place evolution operations in Oracle Database 11g.

Oracle Database 11g uses two different approaches to XML Schema evolution [3]:

- Copy-based approach – all stored documents valid against the old schema are reinserted into the evolved storage; this approach is slower, but there are no restrictions on the type of changes

- In-place evolution – changes in the schema are applied to update the storage, but the stored documents are not reinserted; this approach is faster, but the new schema must be backwards-compatible and the set of allowed evolution operations is limited

The previous section summarized the backwards-compatible changes between the two versions of an XSem-H diagram. When the evolved diagram is translated into an XSD, this new XSD can be used for in-place evolution. However, not all types of changes are supported by Oracle in-place evolution. Table 5.2 shows a list of all operations on the XSD and their counterpart changes in XSem-H diagrams as described in the previous section, in some cases, with other applicability conditions. If all changes in $C_{\mathcal{D},\mathcal{D}',v,v'}$ are instances of changes in Table 5.2 and all applicability conditions are met, in-place evolution approach can be used

XSD operation	XSem-H change
Add an optional element/attribute to a complex type or group/attribute group	*attributeAdded* (5.8) *subordinateComponentAdded* (5.8)
Convert an element from a simple type to a complex type with simple content	no counterpart in XSem-Evo
Increase the value attribute of an existing maxLength element	*attributeTypeChange* (5.5)[1]
Add an enumeration value (only to the end of an enumeration list)	*attributeTypeChange* (5.5)[2]
Add a global element	*diagramRootAdded* (5.3)
Add a global attribute	no counterpart in XSem-Evo
Add or delete a global complex type	*diagramRootAdded* (5.3) *diagramRootAdded* (5.4) *diagramRootRemoved* (5.4)[3]
Add or delete a global simple type	no counterpart in XSem-Evo
Decrease minOccurs/increase maxOccurs attribute value (increase maxOccurs only for data stored as binary XML)	*attrbiuteCardinalityChange* (5.6) *associationCardinalityChange* (5.6)
Add or delete a global group or an attributeGroup	*diagramRootAdded* (5.4) *diagramRootRemoved* (5.4)[4]
Add, modify, or delete a comment or processing instruction	no counterpart in XSem-Evo
Change the xdb:defaultTable attribute value	no counterpart in XSem-Evo

1. Both the old and the new type must be the same types except for the increased `maxLength`.
2. Both the old (T) and the new type (T') must be enumeration types and the list of values of T must be a prefix of the list of values of T'.
3. Only if the removed root or any part of its subtree is not referenced from any structural representative in the diagram.
4. Only if the removed root or any part of its subtree is not referenced from any structural representative in the diagram.

Table 5.2: In-place Evolution Operations and their Counterparts in XSem-Evo

5.6 Detection Algorithm

Possible changes that can occur between two versions of a PSM diagram were all defined in the previous section. The detection algorithm – see Algorithm 2 – looks for combinations of parameters satisfying the change predicates.

The output of Algorithm 2 is $C_{\mathcal{D},\mathcal{D}',v,v'}$ – set of combinations of parameters

satisfying a change predicate.

Algorithm 2 Change Detection

Input: old and new version $v, v' \in \mathcal{V}$, XSem diagram $\mathcal{D}$, new version of the XSem diagram $\mathcal{D}'$

Output: $C_{\mathcal{D},\mathcal{D}',v,v'}$ – set of changes between the two versions of the diagram $\mathcal{D}$.

```
1: C_{D,D',v,v'} ← ∅
2: for all N in D'.N ∪ A_psm ∪ Att_psm do
3:   for all scope type s of all scopes of N do
4:     for all change type c of changes_s do
5:       for all combinations of parameters c̃ satisfying detected change of
         type c on N do
6:         C_{D,D',v,v'} ← C_{D,D',v,v'} ∪ {c̃}
7:       end for
8:     end for
9:   end for
10: end for
```

Invariants for the Set of Changes

Definition 5.6.1 (Subsets of $C_{\mathcal{D},\mathcal{D}',v,v'}$)**.** *We will denote* $C^{addition}_{\mathcal{D},\mathcal{D}',v,v'}$, $C^{removal}_{\mathcal{D},\mathcal{D}',v,v'}$, $C^{migratory}_{\mathcal{D},\mathcal{D}',v,v'} \subseteq C_{\mathcal{D},\mathcal{D}',v,v'}$ *the sets of detected addition, removal and migratory changes. Each of these sets can be factorized by the construct the change addds/removes/-moves:*

$$Add_{e'} = \{\tilde{c} \in C^{addition}_{\mathcal{D},\mathcal{D}',v,v'} : \tilde{c} \textit{ adds construct } e'\}$$
$$Rem_{e} = \{\tilde{c} \in C^{removal}_{\mathcal{D},\mathcal{D}',v,v'} : \tilde{c} \textit{ removes construct } e\}$$
$$Mov_{e'} = \{\tilde{c} \in C^{migratory}_{\mathcal{D},\mathcal{D}',v,v'} : \tilde{c} \textit{ moves construct } e'\}$$
$$Sed_{e'} = \{\tilde{c} \in C^{sedentary}_{\mathcal{D},\mathcal{D}',v,v'} : \tilde{c} \textit{ is a sedentary change of construct } e'\}$$

The following conditions hold for the detected set of changes $C_{\mathcal{D},\mathcal{D}',v,v'}$:

$$
\begin{aligned}
&\forall e \in \mathcal{M}_{psm} : \\
&\qquad |Add_{e'}| \leq 1 \wedge |Rem_e| \leq 1 \wedge |Mov_{e'}| \leq 1 \wedge \\
&\qquad |Add_{e'}| + |Rem_e| + |Mov_{e'}| \leq 1 \wedge \\
&\qquad |Sed_{e'}| + |Mov_{e'}| = 0 \vee |Add_{e'}| + |Rem_e| = 0 \wedge \\
&\qquad |Sed_{e'}| + |Mov_{e'}| + |Rem_e| > 0 \rightarrow getInVer(e', v) \neq null \wedge \\
&\qquad |Sed_{e'}| + |Mov_{e'}| > 0 \rightarrow getInVer(e', v') \neq null \wedge \\
&\qquad |Rem_e| > 0 \rightarrow getInVer(e, v') = null \wedge \\
&\qquad |Add_{e'}| > 0 \rightarrow getInVer(e', v) = null \wedge
\end{aligned}
$$

Chapter 6

Revalidation

This chapter describes an algorithm producing a revalidation script that produces a revalidated document $D' \in \mathcal{S}(\mathcal{D}')$ when applied on XML document $D \in \mathcal{S}(\mathcal{D})$.

In the previous chapter we defined how to obtain the set of changes $C_{\mathcal{D},\mathcal{D}',v,v'}$. Because changes are defined as changes in the XSem model, similarly as XSem model can be translated to an XML schema in any XML schema language, the set of changes can be used to generate a revalidation script in any kind of implementation language.

Assuming that the XQuery Update Facility is the implementation language, each change $\tilde{c} \in C_{\mathcal{D},\mathcal{D}',v,v'}$ would be translated to an XQuery Update command(s):

- addition changes to `insert` commands
- removal changes to `delete` commands
- migratory changes would generate first `insert` command referencing some part of the document and thus copying the content and `delete` command to remove the content from its old location
- sedentary changes would generate `rename` command or again `insert` or `delete` commands

Each command would then be executed upon the revalidated document. The procedure when using DOM API would be analogous.

XSem-Evo uses XSL stylesheets as implementation language due to the wide support for XSLT among the tools working with XML data and especially the database systems supporting XML Schema evolution.

6.1 XSL Specifics

The procedure described in the previous part is not directly applicable when generating revalidation XSL stylesheet for the following reasons:

No Removal XSL does not have any means of explicit removing a content from a document. Removal is achieved by not putting the particular part of content to the output. This is achieved either by the composing the instructions so that the processor never reaches the particular part of content, or by letting the processor go through the content without sending anything to the output.

Processing of Unchanged Content When DOM or XQuery Update Facility is used, an unchanged content does not need to be processed – it will just remain in the document. On the other hand, XSLT must process all content that should be sent to the output.

Output Definitiveness When XSLT processor sends a content to the output, it can not be changed during the same transformation. With XQuery or DOM, several separate changes can be conducted on the same nodes. With XSLT the changes have to be grouped and conducted together.

6.2 Nodes Categorization

Definition 6.2.1 (Imprint changes)**.** *Migratory changes that move a construct from one node to another node, attribute cardinality change and changes that model renaming an XML node will be called* imprint changes*. For imprint change $\tilde{c}$ we define the* affected node *in the following table and denote it $\tilde{c}.affected$. We will say $\tilde{c}$ is an imprint change for a construct N if $\tilde{c}.affected = N$. Table 6.1 lists all types of the imprint with their respective affected nodes.*

From now on we will consider PSM attributes in attribute containers as nodes because they represent XML elements. In the following text, let v and v' be an old version and new version in the model, and $\mathcal{D}'$ new version of diagram $\mathcal{D}$. Considering the features of XSL, we will divide the nodes and attributes in the diagram $\mathcal{D}'$ into these subsets of $\mathcal{D}'.\mathcal{N} \cup \mathcal{A}tt_{psm}$:
$\mathcal{D}'_{base}, \mathcal{D}'_{red}, \mathcal{D}'_{blue}, \mathcal{D}'_{green}, \mathcal{D}'_{group} \subseteq \mathcal{D}'.\mathcal{N} \cup \mathcal{A}tt_{psm}$.

Change type	Affected node
classMovedToClassUnion	former parent association/class union
classMovedOutOfClassUnion	former parent association/class union
attributeMoved (to attribute container)	former parent class/attribute container
attributeCardinalityChange	parent class/attribute container
attributeIndexChange	parent class/attribute container
classElementLabelRemoved	parent association/class union
classElementLabelAdded	parent association/class union
classElementLabelChanged	parent association/class union
contentContainerNameChange	parent superordinate node
subordinateComponentIndexChange	parent superordinate node
subordinateComponentMovedChange	parent superordinate node

Table 6.1: Imprint Changes

Definition 6.2.2 (Group Nodes)**.** *The set* $\mathcal{D}'_{group}$ *is defined as:*

$$\mathcal{D}'_{group} = \{C' \in \mathcal{C}_{psm} | getInVer(C', v) = null \wedge C'.label = null \vee getInVer(C', v).label = null\}$$

Elements of $\mathcal{D}'_{group}$ *will be called* group nodes*.*

Definition 6.2.3 (Significant Nodes, Base Nodes)**.** *Let* $\mathcal{D}'_{sign}$ *be the set of all significant nodes in the diagram.* $\mathcal{D}'_{base}$ *is defined as:*

$$\mathcal{D}'_{base} = \mathcal{D}'_{sign} \cup \mathcal{D}'_{group}$$

Elements of $\mathcal{D}'_{base}$ *will be called* base nodes*.*

Definition 6.2.4 ($C_{\mathcal{D},\mathcal{D}',v,v'}[N]$)**.** $C_{\mathcal{D},\mathcal{D}',v,v'}[N] \subseteq C_{\mathcal{D},\mathcal{D}',v,v'}$ *is the set of changes detected for node* $N \in \mathcal{M}_{psm}$ *and is defined as:*

$$C_{\mathcal{D},\mathcal{D}',v,v'}[N] = \{\tilde{c} \in C_{\mathcal{D},\mathcal{D}',v,v'} | \tilde{c} \text{ is detected for } N\} \cup \{\tilde{c} \in C_{\mathcal{D},\mathcal{D}',v,v'} | \tilde{c} \text{ is imprint change for } N\}$$

Notion $C^{base}_{\mathcal{D},\mathcal{D}',v,v'}[N]$ *will be used for the set of all changes belonging to a base node N.*

$$C^{base}_{\mathcal{D},\mathcal{D}',v,v'}[N] = \bigcup_{N':CSN(N')=N} C_{\mathcal{D},\mathcal{D}',v,v'}[N]$$

Definition 6.2.5 (Green Nodes)**.** *The set* $\mathcal{D}'_{green} \subseteq \mathcal{D}'_{base}$ *is the set of those base nodes that were not changed between versions* v *and* v' *and neither was any node in their subtree.*

$$\mathcal{D}'_{green} = \{M' \in \mathcal{D}'_{base} \mid \forall N' \in \widehat{M'} : C^{base}_{\mathcal{D},\mathcal{D}',v,v'}[N] = \emptyset\}$$

Elements of $\mathcal{D}'_{green}$ *will be called* green nodes*.*

Definition 6.2.6 (Red Nodes)**.** *The set* $\mathcal{D}'_{red} \subseteq \mathcal{D}'_{base}$ *is the set of those base nodes that were changed between versions* v *and* v'*.*

$$\mathcal{D}'_{red} = \{N' \in \mathcal{D}'_{base} \mid C^{base}_{\mathcal{D},\mathcal{D}',v,v'}[N] \neq \emptyset\}$$

Elements of $\mathcal{D}'_{red}$ *will be called* red nodes*.*

Definition 6.2.7 (Blue Nodes)**.** *The set* $\mathcal{D}'_{blue} \subseteq \mathcal{D}'_{base}$ *is the set of those base nodes that were not changed between versions* v *and* v' *but some of their descendant nodes were.*

$$\mathcal{D}'_{blue} = \{M' \in \mathcal{D}'_{base} \mid C^{base}_{\mathcal{D},\mathcal{D}',v,v'}[M'] = \emptyset \wedge \exists N' \in \widehat{M'} : C^{base}_{\mathcal{D},\mathcal{D}',v,v'}[N] \neq \emptyset\}$$

Elements of $\mathcal{D}'_{blue}$ *will be called* blue nodes*.*

The following conditions join the previous definitions:

$$\mathcal{D}'_{base} = \mathcal{D}'_{green} \cup \mathcal{D}'_{blue} \cup \mathcal{D}'_{red}$$
$$\mathcal{D}'_{green} \cap \mathcal{D}'_{blue} = \mathcal{D}'_{red} \cap \mathcal{D}'_{blue} = \mathcal{D}'_{green} \cap \mathcal{D}'_{red} = \emptyset$$

6.3 Revalidation Script Overview

In XSL, as in other languages, stylesheets producing the same output can be written in several forms. To keep it transparent, comprehensible and easily modifiable, the revalidation stylesheet generated by XSem-Evo takes the following form:

- It is a one-pass stylesheet.
- It follows the *navigational stylesheet* pattern described in [15]. It is output oriented and relies on a detailed knowledge of the input document. XPath expressions used for `name` and `match` attributes of all top-level templates are always absolute.
- A top-level template is created for each *red node*.
- Each top-level template describes attributes and direct subelements of the processed red node.
- One common top-level template is added to process all *green* nodes and another to process all *blue nodes*.
- Implicit XSLT stylesheets are never used, because they do not serve the desired purpose.
- The stylesheet grows (counting the number of top-level templates) with the amount of changes made in the diagram, not with the complexity of the diagram.

6.4 XPath Expressions for XSem-H Nodes

In this section we define an XPath expression for each XSem-H construct in a diagram $\mathcal{D}$.

Definition 6.4.1 ($T.XPath$)**.** *For every XSem-H construct T in diagram $\mathcal{D}$ we define an XPath expression $T.XPath$ as :*

$$T.XPath = \begin{cases} T.Parent.XPath \\ \qquad \textit{if } T \in \mathcal{A}_{psm} \cup \mathcal{AC} \cup \mathcal{CU} \cup \mathcal{CH} \\ T.Container.XPath \texttt{ + "/" + } T.aliasOrName \\ \qquad \textit{if } T \in \mathcal{A}tt_{psm} \wedge T.Container \in \mathcal{AC} \\ T.Container.XPath \texttt{ + "/@" + } T.aliasOrName \\ \qquad \textit{if } T \in \mathcal{A}tt_{psm} \wedge T.Container \in \mathcal{C}_{psm} \\ T.Parent.XPath \texttt{ + "/" + } T.n \\ \qquad \textit{if } T \in \mathcal{CC} \wedge T.Parent \neq null \end{cases}$$

$$N.XPath = \begin{cases} \texttt{"/"} + T.n \\ \qquad \textit{if } T \in \mathcal{CC} \wedge T.Parent = null \\ \texttt{"/"} + T.label \\ \qquad \textit{if } T \in \mathcal{C}_{psm} \wedge T.label \neq null \wedge T \in \mathcal{D}.roots \\ T.Parent.XPath + \texttt{"/"} + T.label \\ \qquad \textit{if } T \in \mathcal{C}_{psm} \wedge T.label \neq null \wedge T \notin \mathcal{D}.roots \\ T.Parent.XPath \\ \qquad \textit{if } T \in \mathcal{C}_{psm} \wedge T.label = null \wedge T \notin \mathcal{D}.roots \\ \texttt{"<virt-root>"} \\ \qquad \textit{if } T \in \mathcal{C}_{psm} \wedge T.label = null \wedge T \in \mathcal{D}.roots \end{cases}$$

The text `"<virt-root>"` *is used as a substitute for a root expression* `"/"` *for root classes without element labels.*

6.5 Green Nodes Processing

Nodes in the set $\mathcal{D}'_{green}$ are to be left unchanged and the same applies for their subtrees. XSLT construct <`xsl:copy-of`> can be used for these subtrees. The whole subtree is copied as is to the output without the need to process the nodes in the subtree by the XSLT processor.

Template 6.1 is used to process all significant green nodes and will be denoted $greenNodesTemplate_{\mathcal{D}'}$. The value of match attribute {green-nodes-paths} is an expression that returns the set of all the instances of the green nodes in the diagram. It takes the form of absolute paths ($N.XPath$ of green node N) joined by the `|` operator, e.g. '`/Customer/Purchase | /Customer/Address`'.

```
<xsl:template match='{green-nodes-paths}'>
  <xsl:copy-of select='.' />
</xsl:template>
```

Template 6.1: Green Nodes Template

6.6 Blue Nodes Processing

Nodes in set $\mathcal{D}'_{blue}$ are unchanged, but for each node $M' \in \mathcal{D}'_{blue}$ there is a least one red node N' in $\widehat{M'}$. This implies that <xsl:copy-of> can not be used for blue nodes, because the XSLT parser would not process the instance of node N'.

Instead of <xsl:copy-of>, <xsl:copy> is used to create the XML element and two diagram-independent templates `copyAttributes` and `processContent` are called. Attributes can be copied, because the script never contains top-level templates processing attributes. Template `processContent` calls apply-templates for all subelements. The complete definition of these two named templates can be found on the attached CD together with other definitions of diagram-independent templates (see Apendix A).

Template 6.2 will be denoted $blueNodesTemplate_{\mathcal{D}'}$ and is used to process all *significant* blue nodes. XPath expression {blue-nodes-paths} is obtained analogously as the expression for green nodes.

```
<xsl:template match="{blue-nodes-paths}">
  <xsl:copy>
    <xsl:call-template name='copyAttributes' />
    <xsl:call-template name='processContent' />
  </xsl:copy>
</xsl:template>
```

Template 6.2: Blue Nodes Template

6.7 Red Nodes Processing – Simplified Diagrams

To start off, we will assume a restricted set of constructs for diagrams $\mathcal{D}$ and $\mathcal{D}'$:

$$\begin{aligned} &\forall N \in \mathcal{D}.\mathcal{N} \cup \mathcal{D}'.\mathcal{N} : N \text{ is PSMClass} \rightarrow \\ &\quad N.label \neq null \wedge N.Represented = null \\ &\{\mathcal{D}.\mathcal{N} \cup \mathcal{D}'.\mathcal{N}\} \cap \{\mathcal{CH} \cup \mathcal{CU}\} = \emptyset \end{aligned} \tag{6.1}$$

I.e. no classes without element labels, no structural representatives, content choices and class unions.

By definition 6.2.3 and due to diagram restriction, all red nodes are significant nodes (PSM classes with element labels, content containers and attributes in

attribute containers). For each red node N', a top level template $nodeTemplate_{N'}$ is created in the stylesheet.

The type of the template is determined by the *state* of node N' – whether it was added in version v' or existed in the version v. If it existed in the previous version, we can expect an instance of N' in the documents $\in \mathcal{S}(\mathcal{D})$. To distinguish these two cases, we will use the following definition:

Definition 6.7.1 (Node State)**.** Node state *of node* $N' \in \mathcal{D}'.\mathcal{N}$ *has one of the following values:*

$$N'.state = \begin{cases} existing & if\ Add_{N'} = \emptyset \Leftrightarrow getInVer(N', v) \neq null \\ added & if\ Add_{N'} \neq \emptyset \Leftrightarrow getInVer(N', v) = null \end{cases}$$

Depending on $N'.state$, $nodeTemplate_{N'}$ will be either:

- $appliedNodeTemplate_{N'}$ (existing nodes) or
- $namedNodeTemplate_{N'}$ (added nodes).

Context of Generator

XSLT generator keeps a limited state information called *context*. Here is the first part of *context* attributes:

- $context.BodyNode \in \mathcal{D}'_{base}$ – base node currently being processed
- *context.ProcessedPath* – XPath expression belonging to the currently processed node (in the old version of the diagram)
- $context.ForceCallable \in \{true, false\}$ – flag indicating, that the *state* of nodes (see Definition 6.7.1) is ignored and even *existing* nodes should be treated as *added*

Relative XPath Context attribute *context.ProcessedPath* is used to obtain relative path from *context.BodyNode* to another node.

Value of *context.ProcessedPath* is a simple XPath expression – it contains only names of elements/attribute delimited by `child` axis e.g. '`/Customer/Purchase/Item/@amount`'. Obtaining a relative XPath for *context.ProcessedPath* and another node is rather straightforward. Common prefix

is stripped and necessary upward steps are added if need be. This operation will be denoted *relativeXPath*(a, *context.ProcessedPath*) where $a \in \mathcal{D}'_{sign}$.

Some examples of *relativeXPath*:

context.ProcessedPath	path to node a	relative path
`/Purchase`	`/Purchase/Item/@amount`	`Item/@amount`
`/customer-info/Customer`	`/customer-info/Address/city`	`../Address/city`

GenValue function In situations where new instance of a PSM attribute a' must be generated, we will expect the existence of a function $genValue(a')$, which returns a correct new value for the generated instance.

6.7.1 Red Node Template – Foundations

For each *existing* node N'_e, template $appliedNodeTemplate_{N'_e}$ with `match` attribute will be created:

```
<xsl:template match='{N_e.XPath}'>
    {
        generate red node template body with
            context.ProcessedPath = N_e.XPath,
            context.BodyNode = N'_e
    }
</xsl:template>
```

Note that the value of `match` attribute, which returns an XPath expression (see Definition 6.4.1) for the *old version* of the node N'_e. It matches the instances of N'_e in the documents valid against the old version.

For *added* node N'_a, a named template $namedNodeTemplate_{N'_a}$ is created. Name of the template must be unique, *name* function is used to return a unique name. To make the script transparent, the returned name indicates the location of the node N'_a in the XSem-H tree e.g. '`Customer-Purchase`':

```
<xsl:template name='{name(N'_a)}'>
    {
        generate red node template body with
            context.ProcessedPath = existingAncestor(N'_a).XPath,
            context.BodyNode = N'_a
    }
</xsl:template>
```

The function *existingAncestor*(N'_a) returns the first node A' on the path from N'_a that existed in the previous version ($getInVersion(A', v) \neq null$). For significant root nodes `"/"` expression is used instead of *existingAncestor*(N'_a). For root classes without element labels the algorithm uses an auxiliary expression `"<virt-root>"`.

The template body is generated as follows:

- if the red node N' is a significant node (which we now assume for restricted diagrams), an XML element `E` modeled by N' is added
- if N' is a PSM attribute inside an attribute container, its value is added, otherwise:
  - attributes of element `E` are added (if there are any)
  - subelements of element `E` are added (if there are any)

6.7.2 Leaf (Attribute) Nodes

For a red node a', that is a PSM attribute inside an attribute container, the whole template *leafNodeTemplate*$_{a'}$ for *existing* node is specified as:

```
<xsl:template match='{a.XPath}'>
    <{a'.aliasOrName}>
        <xsl:value-of
            select='{convx_a'(relativeXPath(a, context.ProcessedPath))}' />
    </{a'.aliasOrName}>
</xsl:template>
```

and for *added* node (or when in force callable context) as:

```
<xsl:template name='{name(a')}'>
    <{a'.aliasOrName}>
        {genValue(a')}
    </{a'.aliasOrName}>
</xsl:template>
```

In the first template, notice the usage of function *relativeXPath* that returns an XPath expression – relative path from the path currently being in *context.ProcessedPath* to the instance of attribute a in the old document.

In the second template, the function *genValue* is used to generate value of attribute a'.

Verification The following changes were defined for attributes (not-invalidating changes are omitted): *attributeAliasOrNameChange*, *attributeTypeChange*, *attributeMoved*, *attributeIndexChange*, *attributeCardinalityChange*, *attributeAdded* and *attributeRemoved*. The migratory changes *attributeMoved* and *attributeIndexChange* and also *attributeCardinalityChange*, *attributeAdded* and *attributeRemoved* are targeted in part of the stylesheet where template $leafNodeTemplate_{a'}$ is called. Thus template $leafNodeTemplate_{a'}$ only deals with *attributeAliasOrNameChange* and *attributeTypeChange*.

- *attributeAliasOrNameChange* is solved by using the $a'.aliasOrName$ for the name of the XML element
- *attributeTypeChange* is solved by applying the revalidation function $convx_{a'}$ that expects an XPath expression returning the value of an attribute and converts it to the new type (when type of a was not changed, i.e. if

  $$\{\tilde{c} \in C_{\mathcal{D},\mathcal{D}',v,v'}[N]a' \mid \tilde{c} \textit{ is attributeTypeChange}\} = \emptyset$$

  $convx_{a'}(xpath)$ can be replaced by $xpath$

6.7.3 Inner Nodes – Gathering XML Elements

To define complete body of a template for a red node N' ($nodeTemplate_{N'}$), the subtree of N' must be examined and all subelements in the subtree that model XML elements will have their counterparts in the template $nodeTemplate_{N'}$.

To gather all the subelements, an auxiliary abstract construct *node element wrapper* and an inherited construct *simple node element* will be used.

Definition 6.7.2 (Node Element Wrapper)**.** Node element wrapper *is an auxiliary construct used by the algorithm to wrap XSem-H diagram nodes modeling XML elements.*

Definition 6.7.3 (Simple Node Element)**.** Simple node element *is a subtype of node element wrapper used to wrap a single significant node.*

$$simple\ node\ element = (Element) \tag{6.2}$$

where $Element \in \mathcal{D}'_{sign}$ *is the wrapped significant node.*

Algorithm 3 shows how a list of *node element wrappers* is obtained for a node. Since we now assume a restricted XSem-H diagrams, the algorithm is simple, but will be extended later.

Algorithm 3 Gather Subelements (1)

```
1  function GetSubtreeElements(N' ∈ D'.N)
2      returns subelements_N' – ordered sequence of node element wrapper s.
3  {
4      subelements_N' ← ∅
5      foreach node C' ∈ child nodes of N'
6          getSubtreeElementsInclRoot(N', var subelements_N')
7      return subelements_N'
8  }
9
10 procedure getSubtreeElementsInclRoot(N' ∈ D'.N, var subelements_N')
11 {
12     if N' ∈ {C' ∈ C_psm | C'.label ≠ null} ∪ CC
13         subelements_N' ← new node element wrapper(N')
14     if N' ∈ AC
15         foreach attribute a' ∈ N'.Att_psm
16             subelements_N' ← new node element wrapper(a')
17 }
```

6.7.4 Inner Nodes – Gathering XML Attributes

The approach for gathering attributes is similar to gathering subelements. An auxiliary abstract construct *node attribute wrapper* and an inherited construct *simple node attribute* will be used.

Definition 6.7.4 (Node Attribute Wrapper)**.** *A* Node Attribute Wrapper *is an auxiliary construct used by the algorithm to wrap XSem-H diagram nodes modeling XML attributes.*

Definition 6.7.5 (Simple Node Attribute)**.** *A* Simple Node Attribute *is a subtype of is a subtype of node attribute wrapper used to wrap a single attribute of a PSM class.*

$$\textit{simple node attribute} = (\textit{Attribute}) \tag{6.3}$$

where $\textit{Attribute} \in \mathcal{A}tt_{psm}$ *is the wrapped attribute.*

Algorithm 4 shows how a list of *node attribute wrapper*s is obtained for a node.

Algorithm 4 Gather Attributes (1)

```
 1 function GetAttributes(N' ∈ D'.N)
 2     returns attributes_N' – ordered sequence of node attribute wrappers.
 3 {
 4     attributes_N' ← ∅
 5     getAttributesUnderNode(N', false, var attributes_N')
 6     return attributes_N'
 7 }
 8
 9 procedure getAttributesUnderNode(N' ∈ D'.N, bool firstCall,
10     var attributes_N')
11 {
12     if (N' ∈ C_psm)
13         if (firstCall or N'.label = null))
14             foreach (attribute a' ∈ N'.Att_psm)
15                 attributes_N' ← new node attribute wrapper(a')
16         else return
17     if (N' ∈ C_psm ∪ CC)
18         foreach (node C' ∈ N'.Sub)
19             getAttributesUnderNode(C', false, var attributes_N')
20 }
```

6.7.5 Inner Node Template Body

For a red inner node N', the template adds one XML element when N' is a significant node. The function *name* returns the appropriate name of the XML element modeled by N'. Then template body has two parts: *attributes section* and *elements section*.

```
// red node template
{ if (N' ∈ D'_sign) }
  <{name(N')}> // for significant nodes, include an opening tag
{ end if }
// attribute section (subroutine is introduced..)
    { process attributes(GetAttributes(N')) }
// element section
    { process elements(GetSubtreeElements(N')) }
{ if (N' ∈ D'_sign) }
  </{name(N')}> // for significant nodes, close the tag
{ end if }

// process attributes subroutine
process attributes(att : list of node attribute wrappers)
{
    foreach node attribute wrapper a_w ∈ att
    {
        if a_w is simple node attribute
            generate attribute reference with
                context.CurrentAttribute = a_w
    }
}

// process elements subroutine
process elements(elm : list of node element wrapper)
{
    foreach node element wrapper n_w ∈ elm
    {
        if n_w is simple node element
            generate element reference with
                context.CurrentElement = n_w
    }
}
```

Template 6.3: Inner Red Node Template

```
{ if (a'.state = added ∧ a'.l ≠ 0) ∨ (context.ForceCallable) } // added attrib.
    <xsl:attribute name='{a'.aliasOrName}'>
        {genValue(a')}
    </xsl:attribute>
{ else if C_{D,D',v,v'}[N]a' = ∅} // attribute was not changed
    <xsl:copy-of select='{relativeXPath(a, context.ProcessedPath)}' />
{ else } // attribute was changed
    { var relXPath_a = relativeXPath(a, context.ProcessedPath) }
    { if a.l = 0 ∧ a'.l = 1 }
        // attribute must be present
        <xsl:choose>
            <xsl:when test='{relXPath_a}'>
                <xsl:attribute name='{a'.aliasOrName}'>
                    <xsl:value-of select='{convx_{a'}(relXPath_a)}' />
                </xsl:attribute>
            </xsl:when>
            <xsl:otherwise>
                <xsl:attribute name='{a'.aliasOrName}'>
                    {genValue(a')}
                </xsl:attribute>
            </xsl:otherwise>
        </xsl:choose>
    { else if a.l = 0 ∧ a'.l = 1 }
        // attribute added only when it was present
        <xsl:if test='{relXPath_a}'>
            <xsl:attribute name='{a'.aliasOrName}'>
                <xsl:value-of select='convx_{a'}(relXPath_a)' />
            </xsl:attribute>
        </xsl:if>
    { else }
        // attribute was present
        <xsl:attribute name='{a'.aliasOrName}'>
            <xsl:value-of select='{convx_{a'}(relXPath_a)}' />
        </xsl:attribute>
    { end if }
{ end if }
```

Template Fragment 6.4: Generating Attribute Reference

Definition 6.7.6 (Function *name*)**.** *Function name is defined as follows:*

$$name : \mathcal{D'}_{sign} \rightarrow \mathcal{L}$$

$$name(N) = \begin{cases} N.label & \text{if } N \in \mathcal{C}_{psm} \\ N.n & \text{if } N \in \mathcal{CC} \\ N.aliasOrName & \text{if } N \in \mathcal{A}tt_{psm} \end{cases}$$

Template Fragment 6.4 contains the definition of **generate attribute reference** subroutine and Template Fragment 6.5 the definition of **generate element reference** subroutine.

Generating Attribute References

Template Fragment 6.4 shows how attribute references are generated.

Subroutine **generate attribute reference** (where a' is used as a shortcut for *context.CurrentAttribute.Attribute* and a as a shortcut for its previous version in the last block) has three main blocks. The first block is used by attributes added in the version v' – new attribute is added with generated value. The second block is used for attributes that were not changed between versions v and v' and can be copied. The last block is used by attributes whose definition changed between versions v and v'.

Verification The following changes were defined for attributes (not-invalidating changes are omitted): *attributeAliasOrNameChange*, *attributeMoved*, *attributeTypeChange*, *attributeCardinalityChange*, *attributeAdded* and *attributeRemoved*. The *attributeMoved* change is solved by using the Gather Attributes algorithm – all attributes relevant for the node are included. The *attributeAdded* change is solved in the first block of the subroutine, *attributeRemoved* can be ignored (Gather Attributes will omit removed attributes). Thus, only *attributeAliasOrNameChange*, *attributeTypeChange* and *attributeCardinalityChange* need to be aimed at:

- *attributeAliasOrNameChange* is solved by using the $a'.aliasOrName$ inside the `<xsl:attribute>` construct.
- *attributeTypeChange* is solved by applying the revalidation function $convx_{a'}$ that expects an XPath expression returning the value of an attribute and

converts it to the new type. When the type of a was not changed, i.e. if

$$\{\tilde{c} \in C_{\mathcal{D},\mathcal{D}',v,v'}[N]a' \mid \tilde{c} \textit{ is } \textit{attributeTypeChange}\} = \emptyset$$

$convx_{a'}(xpath)$ can be replaced by $xpath$.

- *attributeCardinalityChange* is solved by checking the cardinality. The only expected cardinality intervals are $(0, 1)$, $(1, 1)$ – an XML attribute can be either optional or mandatory.

Inner Node Elements

Template Fragment 6.5 shows how attribute references are generated.

Element Reference Subroutine **generate element reference** (where N' is used as a shortcut for *context.CurrentElement.Element*) distinguishes two cases. The first one proceeds to generating code that handles cardinality changes and iterated additions (when new construct is added with cardinality > 1). The second one solves cases where cardinality of element N' need not to be dealt with. This happens when either the cardinality did not change from the previous version or the element was added with upper cardinality 1.

```
{ if lowerElementCardinality(N') = 0 ∧
     (N'.state = added ∨ context.ForceCallable) }
     exit; // an optional element need not to be added
{ end if }
{ if (N'.state = existing ∧ cardinalityChanged(N')) ∨
       (N'.state = added ∧ lowerElementCardinality(N') > 1) ∨
       (context.ForceCallable ∧ lowerElementCardinality(N') > 1) }
       generate element cardinality reference
{ else }
       generate element single reference
{ end if }
```

Template Fragment 6.5: Generating Element Reference

Element Single Reference Subroutine **generate element single reference** in Template Fragment 6.6 calls $forceCallableNodeTemplate_{N'}$ if in force callable mode (*context.ForceCallable*). If the called template $forceCallableNodeTemplate_{N'}$ was not generated yet, it is generated now. If not in force callable mode, the template follows the state of node N'.

For newly added nodes $namedNodeTemplate_{N'}$ is called. Thanks to nodes categorization, we can be sure that $namedNodeTemplate_{N'}$ will be a part of the resulting stylesheet because (added nodes are always red nodes and named templates are always generated for added nodes).

For existing nodes, <xsl:apply-templates> is called and the expression that is the value of `select` attribute will either match against $appliedNodeTemplate_{N'}$ (if N' is a red node) or $greenNodesTemplate_{\mathcal{D}'}$ (if N' is a green node) or $blueNodesTemplate_{\mathcal{D}'}$ (if N' is a blue node). The optional `condition` parameter of the routine can restrict the set of the processed existing nodes. This parameter is not used by **generate element reference**, but will be used later.

```
{ parameter: condition (XPath expression, optional)
{ if context.ForceCallable }
    { if forceCallableNodeTemplate_N' not generated yet
        generate forceCallableNodeTemplate_N' }
    <xsl:call-template name='{forceCallableNodeTemplate_N'}' />
{ else }
    { if N'.state = added }
        <xsl:call-template name='{namedNodeTemplate_N'}' />
    { else }
        { var xpath ← relativeXPath(N', context.ProcessedPath) }
        { if condition is set }
            <xsl:apply-templates select='xpath[condition]' />
        { else }
            <xsl:apply-templates select='xpath' />
        { end if }
    { end if }
{ end if }
```

Template Fragment 6.6: Generating Element Single Reference

Force Callable Templates Force callable templates are created in those cases, where a node N' is an existing node (thus $appliedNodeTemplate_{N'}$ will be created for it), but also a named template creating new content without referencing the existing one is needed.

The generating routine is identical (starts at Template 6.3), but the control flow is overridden in appropriate branches by *context.ForceCallable* flag.

Force callable templates generate default instance of an XSem-H subtree. If this is not the desired behavior, the user can replace the content of force callable templates in the script with custom code.

In the examples, force callable templates will have "`-FC`" suffix.

Element Cardinality Reference Subroutine **generate element cardinality reference** in Template Fragment 6.7 deals with cardinality of node N'.

The first part concentrates on instances already present in the document (and is therefore skipped for *added* elements or when in force callable mode). Existing instances are processed again by the *single reference* subroutine – either all existing instances (when the upper cardinality of node N was not decreased i.e. all existing instances can remain in the document) or the first k instances, where k is the new upper cardinality. The `condition` parameter of *single reference* subroutine with built-in XPath function `position` is utilized to restrict the number of instances processed.

The purpose of the second part is to add new instances of N' to the document. Adding several instances may be needed for two reasons:

- N' is a new node and its lower cardinality is > 1
- the lower cardinality of N' changed from l to $l' > l$.

New instances are again created via calling the template $forceCallableNodeTemplate_{N'}$ (which is now generated if it was not generated already). Using the XSLT built-in function `count`, the existing instances are counted (for existing nodes). For added nodes, the generating template must be called k times, k being the lower cardinality of the added node. For nodes where cardinality has changed, the difference between the count of the existing instances and the lower cardinality of N' is used.

```
/* routine called either when cardinality of element N changed
   or N' was added with lower cardinality > 1 */

{ if ¬context.ForceCallable ∧ N'.state = existing
    // cardinality of N' changed, deal with existing nodes
    var c̃ ← getCardinalityChange(N', C_{D,D',v,v'}[N])
    if ¬(c̃ may require deleting)
        generate element single reference
    else
        generate element single reference with
            condition = 'position() ≤ c̃.u'
    end if
  end if
  if context.ForceCallable ∨ (N'.state = added ∧
     getCardinalityChange(N', C_{D,D',v,v'}[N]) may require generating)
     // new nodes need to be created

     if forceCallableNodeTemplate_{N'} not generated yet
        generate forceCallableNodeTemplate_{N'}

     var countExpr
     var lower ← lowerElementCardinality(N')
     if (N'.state = added ∧ context.ForceCallable)
        countExpr ← lower
     else
        var existing ← relativeXPath(N', context.ProcessedPath)
        countExpr ← '{lower} - count({existing})' }
        <xsl:for-each select='1 to {countExpr}'>
            <xsl:call-template
                name='{forceCallableNodeTemplate_{N'}}' />
        </xsl:for-each>
   { end if }
{ end if }
```

Template Fragment 6.7: Generating Element Cardinality Reference

Verification Now we will show how the templates defined in this section revalidate changes defined in Chapter 5 (we still assume a restricted XSem-H diagram defined in Section 6.7). The following changes relate to this part of the algorithm (non-invalidating changes are omitted): *diagramRootRemoved*, *subordinateComponentIndexChange*, *subordinateComponentMoved*, *subordinateComponentAdded*, *subordinateComponentRemoved*, *classElementLabelChanged*, *contentContainerNameChange*, *associationCardinalityChange*, *associationChildAdded*,

associationChildRemoved, *classMovedToAnotherAssociation*, *attributeMoved*, *attributeCardinalityChange*, *attributeAdded*, *attributeRemoved*, *attributeIndexChange* (in this part we are concerned only with attribute changes for attributes in attribute containers).

Gather Subelements algorithm (Algorithm 3) ignores removed components of superordinate nodes (thus *subordinateComponentRemoved*, *associationChildRemoved*, *diagramRootRemoved* and *attributeRemoved* is solved). It also includes newly added nodes, which solves *subordinateComponentAdded*, *associationChildAdded* and *attributeAdded*. New components/attributes have state = *added* and new content is generated for them if necessary. All subelements are returned in the correct order and their references in the inner node template (Template 6.3) are created in the same order, so *subordinateComponentIndexChange* and *attributeIndexChange* are solved.

- *classElementLabelChanged*/*contentContainerNameChange* is solved by using the *name* to name the XML elements.
- *attributeMoved*, *subordinateComponentMoved* and *classMovedToAnotherAssociation* changes are solved by Gather Subelements algorithm (which will include them among the sequence of subelements) and using the *relativeXPath* function that will locate them in the old document.
- *associationCardinalityChange*/*attributeCardinalityChange* are solved in the generate cardinality element reference subroutine.

6.7.6 Example

To conclude the first part of the revalidation algorithm, we will show an example of an evolved diagram and a revalidation stylesheet. Figures 6.1 and 6.2 show the old and new version of the diagram.

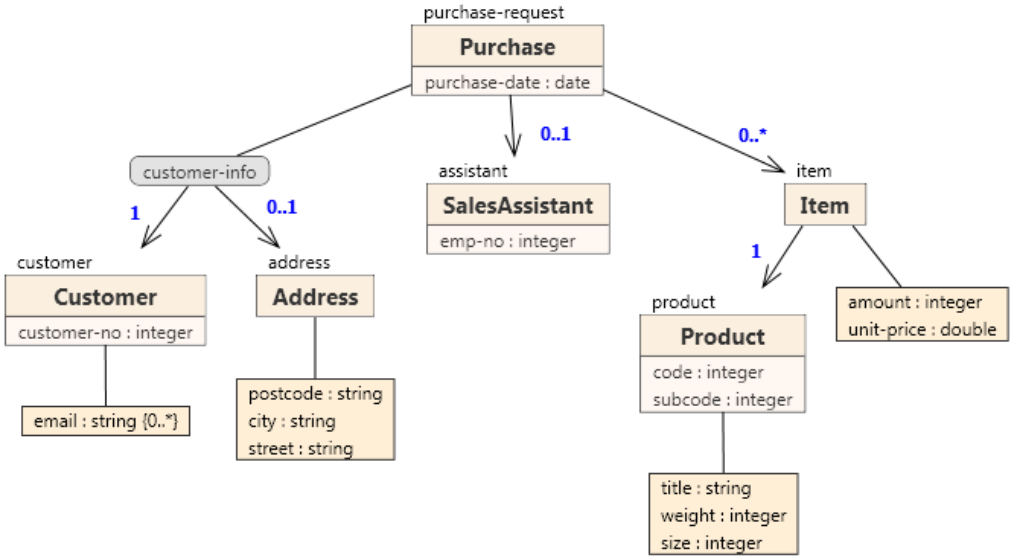


Figure 6.1: Evolution Example 1 - Old Version

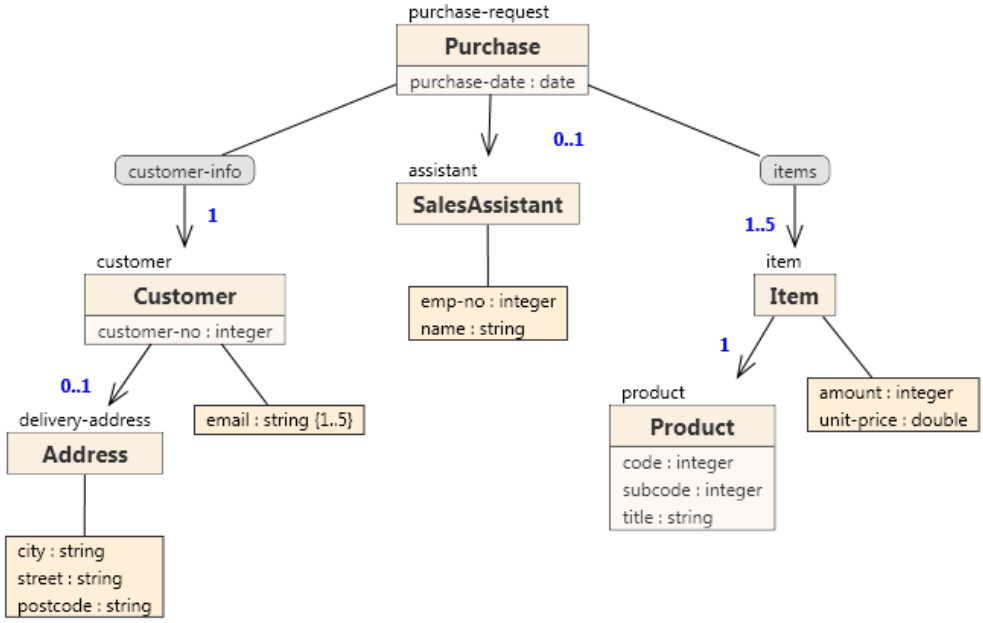


Figure 6.2: Evolution Example 2 - Evolved Version

These are the changes performed between version v and v'

- *subordinateComponentAdded (Purchase, items)*
- *subordinateComponentRemoved (customer-info, customer-info → Address), subordinate-ComponentAdded (customer, association Customer → Address)*
- *classMovedToAnotherAssociation (Address, Customer → Address)* – subelement `address` of element `customer` is now replaced by subelement `delivery-address` of element `customer`.
- *classElementLabelChanged (Address, 'delivery-address')*
- *attributeIndexChange (city, 0), attributeIndexChange (street, 1), attributeIndexChange (postcode, 2)* – attributes were reordered in the attribute container
- *subordinateComponentIndexChange (attribute container in Customer, 1)* – as a result of adding new component preceding the attribute container
- *attributeCardinalityChange (email, 1, 5)* – email was made mandatory (it was optional in previous version), but the total count of emails is now limited to 5
- *subordinateComponentAdded (attribute container in SalesAssistant*
- *attributeAdded (attribute container in SalesAssistant, name, 0)* – new attribute, not present in the previous version
- *attributeMoved (attribute container in SalesAssistant, emp-no, 1)* – attribute was moved from class to an attribute container – in the new version, the instance will be an XML element instead of XML attribute
- *associationCardinalityChange (Purchase-Item, 1, 5)*
- *subordinateComponentMoved (Item, Purchase to items)* – list of purchased items is now wrapped in `items` element
- *subordinateComponentRemoved (Product, attribute container in Product)* – attribute container removed together with attributes `weight` and `size`; attribute `title` was moved to the class Product and will represent XML attribute in the new version
- *attributeMoved (title, Product, 2)*

Stylesheets 6.8, 6.9, 6.10 and 6.11 contain the revalidation stylesheet.

The first part of the revalidation stylesheet deals with changes in the left subtree of class `Purchase`. It relocates content modeled by `Address` and adds element `email` when needed. Force callable template is generated for `email` attribute, which generates an empty element `email`.

Template *appliedNodeTemplate*$_{Customer}$ (`match='/purchase-request/customer-info/customer'`) also restricts the list of email to the maximal length of 5 (using `position` function).

```
<xsl:template match='/purchase-request'>
  <purchase-request>
    <xsl:call-template name='copyAttributes' />
    <xsl:apply-templates select='customer-info' />
    <xsl:apply-templates select='assistant' />
    <xsl:call-template name='purchase-request-items' />
  </purchase-request>
</xsl:template>
<xsl:template match='/purchase-request/customer-info'>
  <customer-info>
    <xsl:apply-templates select='customer' />
  </customer-info>
</xsl:template>
<xsl:template
     match='/purchase-request/customer-info/customer'>
  <customer>
    <xsl:call-template name='copyAttributes' />
    <xsl:call-template name='customer-delivery-address' />
    <xsl:copy-of select='email[position() &lt;= 5]' />
    <xsl:for-each select='1 to 1 - count(email)'>
      <xsl:call-template name='customer-email-FC' />
    </xsl:for-each>
  </customer>
</xsl:template>
<xsl:template name='customer-email-FC'>
  <email></email>
</xsl:template>
<xsl:template name='customer-delivery-address'>
  <delivery-address>
    <xsl:apply-templates select='../address/city' />
    <xsl:apply-templates select='../address/street' />
    <xsl:apply-templates select='../address/postcode' />
  </delivery-address>
</xsl:template>
```

Stylesheet 6.8: Revalidation Stylesheet for Example 1 – Part 1

The second part of the revalidation stylesheet deals with the changes in the middle subtree of class `Purchase`. It converts instances of PSM attribute `emp-no` to XML elements and also adds element `name` (empty).

```
<xsl:template match='/purchase-request/assistant'>
  <assistant>
    <xsl:apply-templates select='@emp-no' />
    <xsl:call-template name='purchase-rq-assistant-name' />
  </assistant>
</xsl:template>
<xsl:template match='/purchase-request/assistant/@emp-no'>
  <emp-no>
    <xsl:value-of select='.' />
  </emp-no>
</xsl:template>
<xsl:template name='purchase-rq-assistant-name'>
  <name></name>
</xsl:template>
```

Stylesheet 6.9: Revalidation Stylesheet for Example 1 – Part 2

The third part (Stylesheet 6.10) shows templates for the blue and green nodes.

```
<!-- blue nodes template blueNodesTemplate_D' -->
<xsl:template match='/purchase-request/item'>
  <xsl:copy>
    <xsl:call-template name='copyAttributes' />
    <xsl:call-template name='processContent' />
  </xsl:copy>
</xsl:template>

<!-- green nodes template greenNodesTemplate_D' -->
<xsl:template
     match='/purchase-request/customer-info/customer/email
      | /purchase-request/item/amount
      | /purchase-request/item/unit-price
      | /purchase-request/customer-info/address/city
      | /purchase-request/customer-info/address/street
      | /purchase-request/customer-info/address/street'>
  <xsl:copy-of select='.' />
</xsl:template>
```

Stylesheet 6.10: Revalidation Stylesheet for Example 1 – Part 3

The last part of the revalidation stylesheet (see Stylesheet 6.11) deals with changes in the right subtree of class `Purchase`.

```
<xsl:template name='purchase-request-items'>
  <items>
    <xsl:apply-templates select='item[position() &lt;= 5]' />
    <xsl:for-each select='1 to 1 - count(item)'>
      <xsl:call-template name='item-FC' />
    </xsl:for-each>
  </items>
</xsl:template>
<xsl:template name='item-FC'>
  <item>
    <xsl:call-template name='item-product-FC' />
    <xsl:call-template name='item-amount-FC' />
    <xsl:call-template name='item-unit-price-FC' />
  </item>
</xsl:template>
<xsl:template name='item-product-FC'>
  <product>
    <xsl:attribute name='code'>0</xsl:attribute>
    <xsl:attribute name='subcode'>0</xsl:attribute>
    <xsl:attribute name='title'></xsl:attribute>
  </product>
</xsl:template>
<xsl:template name='item-amount-FC'>
  <amount>0</amount>
</xsl:template>
<xsl:template name='item-unit-price-FC'>
  <unit-price>0</unit-price>
</xsl:template>

<xsl:template match='/purchase-request/item/product'>
  <product>
    <xsl:copy-of select='@code' />
    <xsl:copy-of select='@subcode' />
    <xsl:attribute name='title'>
      <xsl:value-of select='title' />
    </xsl:attribute>
  </product>
</xsl:template>
```

Stylesheet 6.11: Revalidation Stylesheet for Example 1 – Part 4

The first template $namedNodeTemplate_{items}$ adds the wrapping `item` element

and restricts the list of items to the maximum length of five and also adds one mandatory item in documents where the list is empty (using force callable templates).

The last template $namedNodeTemplate_{product}$ converts `title` from element to attribute.

6.8 Group Nodes

Up to now we only considered classes with element labels. Allowing classes without element labels requires introduction of *group nodes processing*. We will start with an example.

6.8.1 Motivational Example

Figure 6.3 shows an example of a diagram where an element `Catalogue` contains a list of books. Each books has three attributes `ISBN`, `name` and `author`. For all the books in the catalogue, these attributes are all sequentially placed directly in the `Catalogue` element. Figure 6.4 shows a revalidation script that would be generated by the algorithm described in the previous sections.

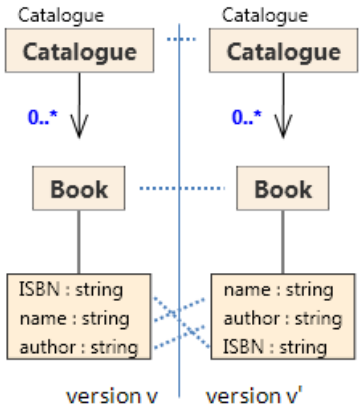


Figure 6.3: Groups Motivational Example

```
<xsl:template match="/Catalogue">
  <Catalogue>
    <xsl:apply-templates select="name" />
    <xsl:apply-templates select="author" />
    <xsl:apply-templates select="ISBN" />
  </Catalogue>
</xsl:template>
```

```
<!-- green nodes -->
<xsl:template match="
   /Catalogue/name
     | /Catalogue/author
   | /Catalogue/ISBN |" >
  <xsl:copy-of select="." />
</xsl:template>
```

Figure 6.4: Groups Motivational Example – Incorrect Script

It is evident that applying this template on any document with more than one instance of `Book` would not have the desired effect, as can be seen in Figure 6.5. It shows document valid against the old version and the same document after applying the template in Figure 6.4.

```
<Catalogue>
  <ISBN>123456789<ISBN>
  <name>Idiot</name>
  <author>F. Dostoyevsky</author>
  <ISBN>987654321<ISBN>
  <name>1984</name>
  <author>G. Orwell</author>
</Catalogue>
```

```
<Catalogue>
  <name>Idiot</name>
  <name>1984</name>
  <author>F. Dostoyevsky</author>
  <author>G. Orwell</author>
  <ISBN>123456789<ISBN>
  <ISBN>987654321<ISBN>
</Catalogue>
```

Figure 6.5: Groups Motivational Example – Effect of revalidation

What happened is that the script violated the separation of the instances of `Book`. To maintain the overall structure of the revalidation script, wee need a mechanism to process only one instance each time inside a template.

For each red group node G' (see Definition 6.2.2), a respective template $groupTemplate_{G'}$ will be created. It will always be a named template and will have one parameter cg (current group).

This template will be called for each instance (using the XSLT construct `xsl:for-each-group`) of the node G' and this instance will be passed to the template as the value of cg. Figure 6.6 shows the correct revalidation template ($greenNodesTemplate_{D'}$ was omitted, it stays the same).

```
<xsl:template match="/Catalogue">
  <Catalogue>
    <xsl:for-each-group select="name | author | ISBN"
      group-starting-with="ISBN">
      <xsl:call-template name="Catalogue-GROUP-Book">
        <xsl:with-param name="cg" select="current-group()" />
      </xsl:call-template>
    </xsl:for-each-group>
  </Catalogue>
</xsl:template>

<xsl:template name="Catalogue-GROUP-Book">
  <xsl:param name="cg" />
  <xsl:apply-templates select="$cg[name() = 'name']" />
  <xsl:apply-templates select="$cg[name() = 'author']" />
  <xsl:apply-templates select="$cg[name() = 'ISBN']" />
</xsl:template>
```

Figure 6.6: Groups Motivational Example – Correct Script

6.8.2 Content Group Nodes Processing

To process group nodes, we will add another type of *node element wrapper* - a *content group*.

Definition 6.8.1 (Content Group)**.** *Construct* content group *is a subtype of node element wrapper used to wrap nodes modeling elements that are children of a content group node.*

$$content\ group = (C, Components)$$

where $C \in \mathcal{D}'_{group}$ *is the group node – PSM Class without an element label – and Components is a sequence of node element wrapper constructs – the grouped elements.*

We can now include this construct into Gather Subelements algorithm (for the previous version see Algorithm 3):

Algorithm 5 Gather Subelements (2)

```
1 function GetSubtreeElements(N' ∈ D'.N)
2     returns subelements_N' – ordered sequence of node element wrapper s.
3     // stays the same
4
5 procedure getSubtreeElementsInclRoot(N' ∈ D'.N, var subelements_N')
6 {
7     if N' ∈ {C' ∈ C_psm | C'.label ≠ null} ∪ CC
8         subelements_N' ← new node element wrapper(N')
9     if N' ∈ AC
10        foreach attribute a' ∈ N'.Att_psm
11            subelements_N' ← new node element wrapper(a')
12    if N ∈ D'_group
13    {
14        var components
15        foreach node C' ∈ child nodes of N'
16            getSubtreeElementsInclRoot(C', var components)
17        subelements_N' ← new content group(N', components)
18    }
19 }
```

6.8.3 Group Node Template

For each node $G' \in \mathcal{D}'_{group} \cap \mathcal{D}'_{red}$, new top-level named template is added to the stylesheet – $groupTemplate_{G'}$, see Template 6.12.

Group templates always have the parameter *$cg* in which the currently processed group is passed. This parameter is later used in all references to nodes laying in the subtree of G.

Notice the assignment of *context.ProcessedPath* in Template 6.12. For node templates, *context.ProcessedPath* was assigned with value $N'.XPath$. In the case of group templates, it is assumed, that the 'currently processed' node is inside the parameter *cg*. For this reason, the value of *context.ProcessedPath* will now end with *$cg*. Parameter *cg* contains a sequence of XML element nodes. Function *relativeXPath* must slightly modified to correctly reference nodes in this sequence.

```
<xsl:template name='{groupName(G′)}'>
    { if ¬context.ForceCallable }
        <xsl:param name='cg' />
    { end if }
    { if G′.label ≠ null }
        <{G′.label}> // include an opening tag
    { end if }
    { process elements(G′.Components) with
        context.ProcessedPath = G′.groupPath
        context.CurrentContentGroup = content group(G′) }
    { if G′.label ≠ null }
        </{G′.label}> // close the wrapping tag
    { end if }
</xsl:template>
```

Template 6.12: Group Node Template

Some examples of *relativeXPath* with content groups (the first example is taken from the motivational example – see Figure 6.3), assuming that element `author` is among the components of the group:

context.ProcessedPath	path to node	relative path
`/Catalogue/$cg`	`/Catalogue/Author`	`$cg[name() = 'Author']`
`/Book/$cg`	`/Book/Author/@ID`	`$cg[name() = 'Author']/@ID`

6.8.4 Referencing Group Templates

Now we can add processing of content groups to the **process elements** subroutine via call to **generate group reference** (processing of simple node elements remains the same). Template 6.13 contains **process elements** subroutine.

Group Reference Subroutine **generate group reference** (Template Fragment 6.14) is similar to **generate element reference**. The first block handles cardinality changes and iterated additions, the second block is designated for those situations, where cardinality does not need to be dealt with. G' is used as a shortcut for *context.CurrentContentGroup.C*.

```
// process elements subroutine
process elements(elm : list of node element wrappers)
{
    foreach node element wrapper n_w ∈ elm
    {
        if n_w is simple node element
            generate element reference with
                context.CurrentElement = n_w
        if n_w is content group
            generate group reference with
                context.CurrentContentGroup = n_w
    }
}
```

Template 6.13: Updated Process Elements Subroutine – Group Nodes

```
{ if lowerElementCardinality(G') = 0 ∧
    (G'.state = added ∨ context.ForceCallable) }
    exit; // optional, group need not to be added
{ end if }
{ if (G'.state = existing ∧ cardinalityChanged(G')) ∨
     (G'.state = added ∧ lowerElementCardinality(G') > 1) ∨
     (context.ForceCallable ∧ lowerElementCardinality(G') > 1 }
     generate group cardinality reference
{ else }
     generate group single reference
{ end if }
```

Template Fragment 6.14: Generating Group Reference

Group Single Reference Subroutine **generate group single reference** in Template Fragment 6.15 again calls $forceCallableGroupTemplate_{G'}$ if in force callable mode.

For newly added groups $groupTemplate_{G'}$ is called (group templates are always named templates). Nothing is passed as 'current group' to *$cg* leaving its value to the default empty sequence, but this is parameter will never be used inside a template for a newly added group.

Group is not invalidated even when some of the nodes below the group have been invalidated, but it is not necessary to process the group as a whole. If the group was not invalidated, `xsl:apply-templates` is used for all the components in group. Function *getGroupPopulation* returns an XPath expression that returns

the set of all nodes in the group. In the motivational example this expression is `"name | author | ISBN"`.

```
{ parameter: condition (XPath expression, optional)
{ if context.ForceCallable }
    // call the force callable template
    { if forceCallableGroupTemplate_G' not generated yet
        generate forceCallableGroupTemplate_G' }
    <xsl:call-template name='{forceCallableGroupTemplate_G'}' />
{ else if G'.state = added }
    // group is new -- just call the group template
    <xsl:call-template name='{groupTemplate_G'}' />
{ else if G' ∉ D'_red ∧ condition is not set
    // group is existing, but not invalidated
    var population = getGroupPopulation(context.CurrentContentGroup) }
    <xsl:apply-templates match='{population}' />
{ else
    var population = getGroupPopulation(context.CurrentContentGroup)
    <xsl:for-each-group select='population'
        {groupDistributingAttribute(context.CurrentContentGroup)} >
      { if condition is set}
          <xsl:if test='{condition}' >
              <xsl:call-template name='{groupTemplate_G'}'
                  <with-param name='cg'
                      select='current-group()' />
              </xsl:call-template>
          </xsl:if>
      { else }
          <xsl:call-template name='{groupTemplate_G'}'
              <with-param name='cg'
                  select='current-group()' />
          </xsl:call-template>
      { end if }
    </xsl:for-each-group>
{ end if }
```

Template Fragment 6.15: Generating Group Single Reference

The last block is used when the group itself is invalidated and each group must be processed separately, but as a whole (e.g. when components of the group are reordered).

XSL `for-each-group` construct is utilized to distribute the subelements in `population` (nodes returned by the XPath expression in the value of `select` attribute) into distinct groups. XSLT provides several ways of determining in which

group each element belongs, each applicable in different situation. Each way is expressed in the form of an attribute of element `for-each-group`. Function *groupDistributingAttribute* returns a string containing the appropriate attribute and its value, (for example `group-starting-with="ISBN"` in the motivational example – see Figure 6.6). The different ways of distributing content to groups will be explained in the next paragraph.

Inside `for-each-group`, function `current-group()` is used to assign the value of parameter *$cg* of the called group template $groupTemplate_{G'}$.

The subroutine again has parameter `condition` which, if specified, is used to restrict the amount of existing instances processed (using XSL `if` construct).

Separating Group Instances In XSL, `for-each-group` can use one of following attributes for separating *population* into groups [15]:

name	value	meaning
`group-by`	XPath expr.	Grouping key. Items with common values for the grouping key are to be allocated to the same group.
`group-adjacent`	XPath expr.	Grouping key. Items with common values for the grouping key are to be allocated to the same group if they are adjacent in the population.
`group-starting-with`	Pattern	A new group will be started for each item in the population that matches this pattern.
`group-ending-with`	Pattern	A new group will be started following an item in the population that matches this pattern.

Function *groupDistributingAttribute* selects the appropriate way to allocate population to groups. Several situations can be identified.

Starts of Groups Construct `group-starting-with` with value that matches the first member F' of *Components* can be used safely when F' is not optional and either (a) upper cardinality of F' is 1 or (b) there is another non-optional member $H' \in Components : H' \neq F'$. In the (b) case, value of `group-starting-with` must be adjusted to start new groups only with the first instance of F'. Figure 6.7 contains examples for different situations.

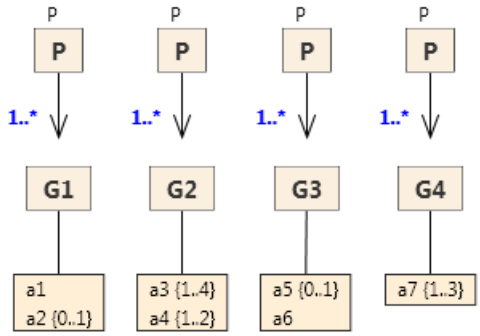


Figure 6.7: Model Examples of Groups

- In the first example, attribute `a1` has cardinality (1,1) and thus can be used to identify starts of groups without complications – *groupDistributingAttribute* would return `group-starting-with="a1"`.
- In the second example, attribute `a3` has cardinality (1,4), but there is another attribute `a4` among components of the group,. Thus the group always starts with an instance of `a3` (and the following instances belong to the same group) – *groupDistributingAttribute* would return `group-starting-with="a3[not(preceding-sibling::a3[1] is preceding-sibling::*[1])]"`.
- For the third example, both the previous approaches fail because the first component – attribute `a5` – is optional. However, this situation can be solved by using an analogous approach with `"group-ending-with=a6"`.
- The last example is solved neither by `group-starting-with` nor with `group-ending-with`, because from the XSem-H model, it is not clear what separates groups from one another.

Ends of Groups Attribute `group-ending-with` can be used in a completely analogous manner to `group-ending-with` with previously stated conditions now applied on the last component in the group.

Size of Groups If the amount of elements in each group is known to be equal for all groups, this information can be exploited when allocating elements

to groups. Instances of group G1 in Figure 6.8 will always contain two elements – one for `a1` or `a2` and `a3`.

In this case, `group-adjacent="(position()-1) idiv 2"` can be a correct result of *groupDistributingAttribute*. In general: `group-adjacent= "(position()-1) idiv` *size*`"` where *size* is the fixed size of each group and `idiv` is an XPath integer division operator.

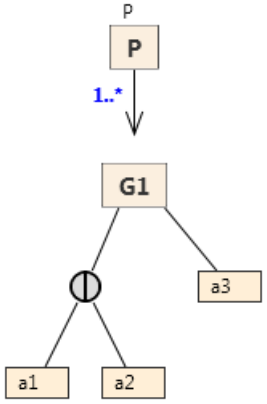


Figure 6.8: Group with Fixed Size

Other Criteria In some cases, the algorithm can not decide how to allocate elements to groups by itself and an input from the user is needed (group `G4` in Figure 6.7 is one example). The user can utilize some internal knowledge of the problem domain (for example some integrity constraints) that is not represented in the XSem-H model.

Group Cardinality Reference Subroutine **generate group cardinality reference** in Template Fragment 6.16 fulfills the same task as **generate element cardinality reference** in the previous section, only for group nodes.

The first part concentrates on instances already present in the document (and is therefore skipped for *added* groups or when in force callable mode). Existing instances are processed again by the *single reference* subroutine – either all existing instances (when the upper cardinality of group G' was not decreased i.e. all existing instances can stay in the document) or the first k instances, where k is the new upper cardinality. The `condition` parameter of *single reference* subroutine with built-in XPath function `position` is utilized to restrict the number

of instances processed.

```
/* routine called either when cardinality of content group
   changed or group was added with lower cardinality > 1 */

{ if ¬context.ForceCallable ∧ G'.state = existing
    // cardinality of G' changed, deal with existing nodes
    var c̃ ← getCardinalityChange(G', C_{D,D',v,v'}[G])
    if ¬(c̃ may require deleting)
        generate group single reference
    else
        generate group single reference with
            condition = 'position() ≤ c̃.u'
    end if
  end if
  if context.ForceCallable ∨ (G'.state = added ∧
    getCardinalityChange(G', C_{D,D',v,v'}[N]) may require generating)
    // new nodes need to be created

    if forceCallableGroupTemplate_{G'} not generated yet
        generate forceCallableGroupTemplate_{G'}

    var countExpr
    var lower ← lowerElementCardinality(N')
    if (N'.state = added ∧ context.ForceCallable)
        countExpr ← lower
    else
        var existing ← countGroupsExpr(G)
        countExpr ← '{lower} - count({existing})' }
        <xsl:for-each select='1 to {countExpr}'>
            <xsl:call-template
                name='{forceCallableGroupTemplate_{N'}}' />
        </xsl:for-each>
   { end if }
{ end if }
```

Template Fragment 6.16: Generating Group Cardinality Reference

The purpose of the second part is again to add new instances of G' to the document.

New instances are again created via calling the force callable template ($forceCallableGroupTemplate_{G'}$). For added groups, the generating template must be called k times, k being the lower cardinality of the added group; for nodes where cardinality has changed, the difference between the number of the

existing instances and the lower cardinality of G' is used. The number of existing instances is obtained via call to *countGroupsExpr* function, which returns an XPath expression.

Counting groups is a problem very similar to separating groups (it is evident that when the algorithm knows how to allocate elements into groups, it can also count them) which as discussed above. When `group-starting-with` (or `group-ending-with`) is used to allocate elements to groups, the same expression as is the value of this attribute (as returned by *groupDistributingAttribute*) function can be used to count existing groups.

6.8.5 Verification

Most of the changes related to group nodes are solved again by the revised Gather Subelements algorithm (Algorithm 5). Group cardinality reference again solves *associationCardinalityChange*, addition and removal changes are solved in the same manner as for simple node elements.

What remains are element label changes:

- *classGivenElementLabel* classes that do not have element label's in version v are group nodes by Definition 6.2.2. As such, group node reference identifies existing instances using `for-each-group` and each instance is wrapped in the new element when executing the template $groupTemplate_{G'}$.
- *classElementLabelChanged* does not involve group nodes and was already discussed for simple node elements.
- *classElementLabelRemoved* does not involve group nodes (class did have an element label in version v therefore it does not belong among group nodes, see Definition 6.2.2). Node template will be generated for such class and the template will remove the wrapping element.

6.9 Content Choices and Class Unions

In this section, we will allow content choices and class unions in the diagram.

First, we have to modify algorithms Gather Subelements/Attributes to cope with these two constructs.

6.9.1 Gathering XML Elements

Up to now, we used *simple node element* and *content group* constructs for all nodes. For content choice/class union nodes, we add abstract *choice base elements wrapper* and specialized *choice elements* and *union elements* auxiliary constructs.

Definition 6.9.1 (Choice Base Elements Wrapper)**.** *Abstract construct* choice base elements wrapper *is a subtype of node element wrapper*

$$choice\ base\ elements\ wrapper = (Options)$$

where Options is a sequence of choice element option constructs.

Definition 6.9.2 (Choice Element Option)**.** *Construct* choice element option *is a subtype of node element wrapper*

$$choice\ element\ option = (Items)$$

where Items is a sequence of node element wrapper constructs.

Definition 6.9.3 (Choice Elements)**.** *Construct* choice elements *is a subtype of choice base elements wrapper*

$$choice\ elements = (CCh, Items)$$

where Items is a sequence of node element wrapper constructs and CCh is a content choice node.

Definition 6.9.4 (Union Elements)**.** *Construct* union elements *is a subtype of choice base elements wrapper .*

$$union\ elements = (CU, Items)$$

where Items is a sequence of node element wrapper constructs and CU is a class union node.

Now we can add new block to Gather Subelements algorithm that collects choice/union nodes:

Algorithm 6 Gather Subelements (3)

```
1 function GetSubtreeElements(N' ∈ D'.𝒩)
2     // stays the same
3
4 procedure getSubtreeElementsInclRoot(N' ∈ D'.𝒩, var subelements_N')
5 {
6     if N' ∈ {C' ∈ C_psm | C'.label ≠ null} ∪ 𝒞𝒞 // simple element
7         subelements_N' ← new node element wrapper(N')
8     if N' ∈ 𝒜𝒞 // simple elements
9         foreach attribute a' ∈ N'.𝒜tt_psm
10            subelements_N' ← new node element wrapper(a')
11    if N ∈ D'_group // groups
12    {
13        var components
14        foreach node C' ∈ child nodes of N'
15            getSubtreeElementsInclRoot(C', var components)
16        subelements_N' ← new content group(N', components)
17    }
18    if N' ∈ 𝒞ℋ ∨ N' ∈ 𝒞𝒰 // choices, unions
19    {
20        var options // each child will become one option
21        foreach node O' ∈ child nodes of N'
22        {
23            var items // each option has its own contents
24            getSubtreeElementsInclRoot(O', var items)
25            if |items| > 0 // empty lists are discarded
26                options ← new choice element option(Items = items)
27        }
28        if |options| > 0 // empty lists are discarded
29            if N' ∈ 𝒞ℋ // content choice
30                subelements_N' ← new choice elements
31                    (CCh = N', Options = options)
32            else // class union
33                subelements_N' ← new union elements
34                    (CU = N', Options = options)
35    }
36 }
```

6.9.2 Gathering XML Attributes

In a similar manner as Gather Subelements, algorithm Gather Attributes also needs to be extended (see Template 6.17).

```
// process attributes subroutine
process attributes(att : list of node attribute wrappers)
{
    foreach node attribute wrapper a_w ∈ att
    {
        if a_w is simple node attribute
            generate attribute reference with
                context.CurrentAttribute = a_w
        else if a_w is choice attributes
            generate choice attributes reference with
                context.CurrentChoiceAttributes = a_w
        else if a_w is union attributes
            generate union attributes reference with
                context.CurrentUnionAttributes = a_w
    }
}

// process elements subroutine
process elements(elm : list of node element wrappers)
{
    foreach node element wrapper n_w ∈ elm
    {
        if n_w is simple node element
            generate element reference with
                context.CurrentElement = n_w
        else if n_w is content group
            generate group reference with
                context.CurrentContentGroup = n_w
        else if n_w is union elements
            generate choice elements reference with
                context.CurrentChoiceElements = n_w
        else if n_w is choice elements
            generate union elements reference with
                context.CurrentUnionElements = n_w
    }
}
```

Template 6.17: Updated Process Elements / Process Attributes Subroutines

Analogously to *choice base elements wrapper*, *choice element option*, *choice*

elements and *union elements* we will use *choice base attributes wrapper*, *choice attribute option*, *choice attributes* and *union attributes*.

6.9.3 Updated Inner Red Node Template

Now we can modify subroutines **process attributes** and **process elements** first introduced in Template 6.3 (**process elements** was already extended once in Template 6.13) – we add processing for the new constructs. The result is Template 6.17

6.9.4 Content Choice Reference

Elements

Template Fragment 6.18 shows how algorithm handles elements under content choice.

```
{ if context.ForceCallable
    if mustHaveInstance(CCh'_n)
        var choiceToGenerate ← getChoiceToGenerate(CCh'_n)
        process elements(choiceToGenerate.Items)
  else if choiceInvalidated(CCh'_n) }
    <xsl:choose>
        { foreach choice element option o ∈ CCh'_n.Options
            var test ← relativeXPath(o.Items, context.ProcessedPath) }
            <xsl:when test='{test}'>
                { process elements(o.Items) }
            </xsl:when>
        { end foreach}
        { if mustHaveInstance(CCh'_n) ∧ canRequireGenerating(CCh'_n) }
            <xsl:otherwise>
                { var choiceToGenerate ← getChoiceToGenerate(CCh'_n)
                  process elements(choiceToGenerate.Items) with
                    context.ForceCallable }
            </xsl:otherwise>
        { end if }
    </xsl:choose>
{ else }
    process not invalidated elements(CCh'_n.Options)
{ end if }
```

Template Fragment 6.18: Generating Content Choice Elements Reference

The first bock is used when in force callable context. Function *mustHaveInstance* returns true if the content choice must have an instance in the document i.e. it is not true that all *options* of the choice are optional[1] . If *mustHaveInstance* returns true, one of the choices is selected and the content for the choice is generated using *process elements* subroutine.

The function *choiceInvalidated* returns true if the choice was modified in such a way that the changes can not be solved at the level of its components (in templates for the respective components). This is necessary when some of the components were removed, the whole choice was added in version v or multiplicity of some of the components have changed.

The middle block generates the full choice template which uses the construct `xsl:choose`. A separate `xsl:when` block is created for each option. The value for `test` attribute is a relative path (obtained by combining the relative paths of respective $CCh'_n.Items$).

The `xsl:otherwise` block is added when a) the content choice must have an instance in the document (*mustHaveInstance* returns *true*) and b) the changes between versions v and v' may require the instance to be generated. This case occurs when for example some of the components of $CCh'_n$was removed – the instance of this component can not be used in the new version, but CCh'_n' must have an instance, thus it must be generated. Another case is when some options were optional in the old version and they are not in the new version. See example in Figure 6.9.

The last block is designated for the situations where no revalidation for the actual choice node is needed (all nodes under the choice node are green or blue nodes).

Attributes

Subroutine for generating **content choice attributes reference** is analogous to **content choice elements reference** (Template Fragment 6.18). The subroutine uses appropriate attribute constructs mentioned in Gathering Attributes (*choice attributes* etc., see p. 141) instead of the constructs defined for elements.

Example

Figure 6.9 contains an example of usage of a content choice.

[1] An example of an optional option is a *simple node element e* where *e.Element* is a PSM attribute (inside an attribute container) with lower cardinality = 0.

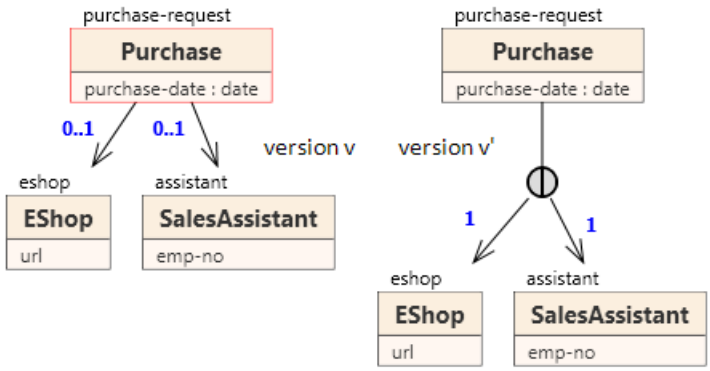


Figure 6.9: Adding a Content Choice

```
<xsl:template match="/purchase-request">
  <purchase-request>
    <xsl:call-template name="copyAttributes" />
    <xsl:choose>
      <xsl:when test="eshop">
        <xsl:apply-templates select="eshop" />
      </xsl:when>
      <xsl:when test="assistant">
        <xsl:apply-templates select="assistant" />
      </xsl:when>
      <xsl:otherwise>
        <xsl:call-template name="eshop-FC" />
      </xsl:otherwise>
    </xsl:choose>
  </purchase-request>
</xsl:template>
```

```
<xsl:template name="eshop-FC">
  <eshop>
    <xsl:attribute name="url"/>
  </eshop>
</xsl:template>

<!--green nodes template-->

<xsl:template match=
  "/purchase-request/eshop
  | /purchase-request/assistant">
  <xsl:copy-of select="." />
</xsl:template>
```

Figure 6.10: Revalidation Script for Diagram in Figure 6.9

The diagram in Figure 6.9 introduces content choice in version v' instead of two optional subtrees in version v. For both options a `when` block is added and the `otherwise` block is added too, because the document must now contain one of the subtrees, but in the old version it could have contained none of them. Thus, if $D \in \mathcal{S}(\mathcal{D})$ contains neither `eshop` nor `assistant`, default instance of the first (`eshop`) subtree is generated.

Verification

This section focused on the changes of the content choice:

- *subordinateComponentAdded*, *subordinateComponentRemoved*, *subordinateComponentIndexChange*, *subordinateComponentMoved* where the content choice is in the role of the subordinate component and the superordinate node is another node
- changes in the collection of components of a content choice (i.e. the same changes, but content choice is now in the role of the superordinate node)

The first set of changes is solved completely by the revised Gather Subelements/Attributes algorithm (see Algorithm 6). This algorithm considers content choices when creating the template for its parent.

Changes in the collection of the components of the content choice will all be handled when the *Options* collection is initialized. Because the order of components of a content choice is not significant, *subordinateComponentIndexChange* can be ignored completely.

6.9.5 Class Union Reference

Elements

Unlike content choice construct, class unions have an association. This association can have arbitrary cardinality interval and thus cardinality changes need to be addressed also with possible iterated generation (similar to *simple node elements* and *content groups*).

Procedure **generate union elements reference** can be found in Template Fragments 6.19, 6.20 and 6.21.

Treatment of union nodes differs for nodes with leading association with upper cardinality > 1 and for class unions containing classes without an element label (content group nodes).

Definition 6.9.5 (Union Complexity)**.** *Auxiliary attribute* complexity *of a class union node CU is defined as:*

$$CU.complexity = \begin{cases} single & \text{if } CU.A.u \not> 1 \\ withoutGroups & \text{if } \forall\ C \in CU.Comp : C.label \neq null \\ withGroups & \text{if } \exists\ C \in CU.Comp : C.label = null \end{cases}$$

Template Fragment 6.19 processes content already existing in the document.

```
{ if (context.ForceCallable ∨ CU'_n.state = added) ∧
     lowerElementCardinality(CU'_n) = 0
     return
  else if ¬context.ForceCallable
     if choiceInvalidated(CU'_n)
        if CU'_n.complexity = single
           generate union choices(false)
        else
           var population = getUnionPopulation(CU'_n)
           var distribution = getUnionDistribution(CU'_n)
           var restrict = cardinalityChanged(CU'_n) ∧
               getCardinalityChange(CU'_n, C_{D,D',v,v'}[CU'_n]) may require deleting
           var l' = lowerElementCardinality(CU'_n) }
         { if CU'_n.complexity = withoutGroups }
             <xsl:for-each select='{population}'>
         { else }
             <xsl:for-each-group select='{population}'
                {distribution} >
         { end if }
            { if restrict }
                <xsl:if test='position() <= {l'}>
            { end if
                   generate union choices(CU'_n.complexity = withGroups)
              if restrict }
                </xsl:if>
            { end if }
         { if CU'_n.complexity = withoutGroups }
              </xsl:for-each>
         { else }
              </xsl:for-each-group>
         { end if
        end if
     else
        process not invalidated elements(
          context.CurrentUnionElements.Options)
     end if
{ end if }
```

Template Fragment 6.19: Class Union – Elements (Part 1.)

In the following text, we will use CU'_n as a shortcut for *context.CurrentUnionElements*. If there are no changes in the collection $CU'_n.Comp$ itself or in the cardinality of any of the components, there is no need for any explicit processing of the class union node and the potential changes

will be solved in the templates of the respective components (**process not invalidated elements** is used again). Otherwise, the union node must be processed.

For unions with *complexity* = *single*, the algorithm can directly proceed to processing of the individual components (**generate union choices** call). Otherwise, instances of respective components are separated first (in order to be counted) using `for-each` or `for-each-group` construct (depending on the *complexity* of the union node) and then the algorithm again proceeds to processing individual components.

If the number of instances is restricted (due to cardinality change in the association leading to the union node), `if` construct and `position` function are used.

Function *getUnionPopulation* returns an XPath expression that covers all the existing instances of all the components in $CCh'_n.Comp$. Function distribution returns an attribute that allocates each component instance into a group (in the same way as group instances are separated, see paragraph Separating Group Instances, p. 134).

```
{ if choiceInvalidated(CU'_n)
    var countExpr
    var distribution ← getUnionDistribution(CU'_n)
    if (distribution not empty)
      countExpr ← '{lower} - count({distribution})'
    else
      countExpr ← '{lower}
    end if
    if forceCallableUnionTemplate_CU'_n not generated yet
      generate forceCallableUnionTemplate_CU'_n
    <xsl:for-each select='1 to {countExpr}'>
      <xsl:call-template name={forceCallableUnionTemplate_CU'_n}>
        { if processing group ∧¬context.ForceCallable }
          <xsl:with-param name='cg' value='$cg' />
        { end if }
      <xsl:call-template/>
    </xsl:for-each>
{ end if }
```

Template Fragment 6.20: Class Union – Elements (Part 2.)

The second part of the template in Template Fragment 6.20 generates content when needed. This may be necessary either when lower cardinality of association

$CCh'_n.A$ is increased or when a component of the union is removed. Removal of a component results in removing of all its instances and thus decreasing the total number of instances in the document.

Force callable template is again used for generating content. Expression returned by *getUnionDistribution* is used to count the existing instances.

```
// generate union choices subroutine
generate union choices(group: bool)
{
  { if group }
    { if CU'_n.complexity = withoutGroups }
      <xsl:variable name='cg' select='current()' />
    { else }
      <xsl:variable name='cg' select='current-group()' />
    { end if }
  { end if }
  <xsl:choose>
    { foreach choice element option o ∈
                    context.CurrentUnionElements.Options
      // always |o.Items| = 1
      var n_w ← o.Items[1]
      if n_w is simple node element
        if getInVer(n_w.Element, v) = null
          continue
        var test = ← relativeXPath(n_w.Element, context.ProcessedPath) }
        <xsl:when test='{test}'>
            { generate element single reference(o.Items) }
        </xsl:when>
      else if n_w is content group
        if getInVer(n_w.C, v) = null
          continue
        var test = ← getGroupPopulation(n_w) }
        <xsl:when test='{test}'>
            { generate group single reference(o.Items) }
        </xsl:when>
      { end if }
    { end foreach }
  </xsl:choose>
}
```

Template Fragment 6.21: Class Union – Elements (Part 3.)

The last part of this template on Template Fragment 6.21 contains definition of **generate union choices**, a subroutine referenced earlier in Tempate

Fragment 6.19.

This subroutine creates `choose` and `when` blocks for the class union and its respective components. In case of class unions (unlike with content choices), options in *Options* collection always have only one *item* which is either a *simple node element* (wrapping class with element label) or *content group* (wrapping class without label).

The added content is skipped and depending on the particular option, either **generate element single reference** or **generate group single reference** is called.

Attributes

Subroutine **generate union attributes reference** is again analogous to **generate union elements reference** (Template Fragment 6.19 – 6.21). The subroutine again uses appropriate attribute constructs mentioned in Gathering Attributes (*choice attributes* etc., see p. 141) instead of the constructs defined for elements.

Example

Figure 6.11 contains an example of usage of a class union, the revalidation script is depicted in Figure 6.12. As we can see, a new class union node was introduced in version v' which encompasses classes `article` and `book`. This effectively allows for interleaving of `article` and `book` elements, whereas version v required all `book` elements after all `article` elements.

After examining the diagram in Figure 6.11, we can see that $\mathcal{S}(\mathcal{D}) \subseteq \mathcal{S}(\mathcal{D}')$, but adding a check to the algorithm that would recognize these particular situations would excessively complicate the algorithm. That is why this is ignored and a union reference is created as if revalidation is needed (but the processing is correct, the revalidation will not change the document).

Revalidation can use `for-each` construct, because the union is without content groups. For both components `article` and `book` a `when` block is created. Since both components are green nodes, their contents can be copied using `copy-of`.

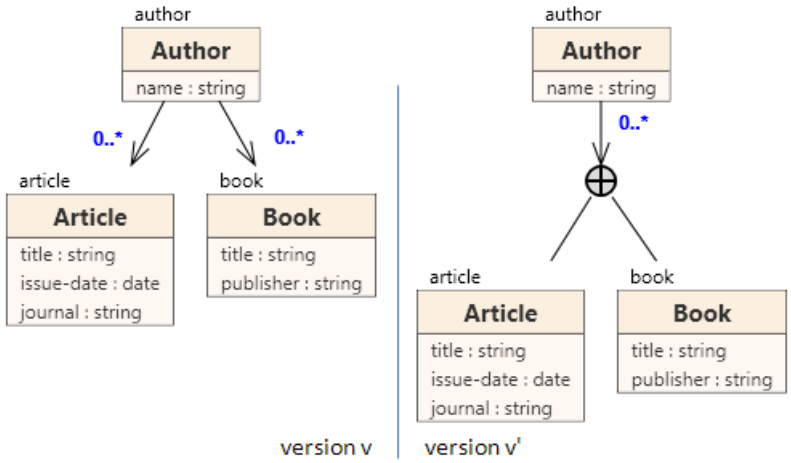


Figure 6.11: Class Union Example

```
<xsl:template match="/author">
  <author>
    <!--Copy attributes-->
    <xsl:call-template name="copyAttributes" />
    <!--Content of Author - 'author'-->
    <xsl:for-each select="article | book">
      <xsl:variable name="cg" select="current()" />
      <xsl:choose>
        <xsl:when test="$cg[name() = 'article']">
          <xsl:copy-of select="$cg[name() = 'article']" />
        </xsl:when>
        <xsl:when test="$cg[name() = 'book']">
          <xsl:copy-of select="$cg[name() = 'book']" />
        </xsl:when>
      </xsl:choose>
    </xsl:for-each>
  </author>
</xsl:template>
<!--green nodes template-->
<xsl:template match="/author/article | /author/book">
  <xsl:copy-of select="." />
</xsl:template>
```

Figure 6.12: Class Union Example – Revalidation Script

Verification

This section focused on the changes of a class union:

- *associationChildAdded*, *associationChildRemoved* and *classUnionMoved* where the node is a class union

- changes in the collection of components of a class union (*classAddedToUnion*, *classRemovedFromUnion*, *classMovedToClassUnion*, *classMovedOutOfClassUnion*, *classIndexChange*)

The first set is of changes is solved completely by the revised Gather Subelements/Attributes algorithm (see Algorithm 6). This algorithm considers class unions when creating the template for its parent.

Changes in the collection of the components of the content choice will all be handled when the *Options* collection is initialized. Because the order of components of a class union is not significant, *classIndexChange* can be ignored completely.

6.10 Structural Representatives

Finally, we add support for structural representatives in the evolved diagram. We again have to extend algorithms Gather Subelements/Attributes to cope with these two constructs.

6.10.1 Gathering XML Elements/Attributes

We add two new auxiliary construct derived from *node element wrapper* resp. *node attribute wrapper*.

Definition 6.10.1 (Structural Representative Elements)**.** *Abstract construct* structural representative elements *is a subtype of node element wrapper*

$$structural\ representative\ elements = (C, Represented)$$

where C is a PSM class and Represented is another PSM class linked to the same PIM class as C and C.Represented = Represented.

Definition 6.10.2 (Structural Representative Attributes)**.** *Abstract construct* structural representative attributes *is a subtype of node attribute wrapper*

$$structural\ representative\ attributes = (C, Represented)$$

where C is a PSM class and Represented is another PSM class linked to the same PIM class as C and C.Represented = Represented.

Content inherited from a represented class can occur in two situations: either (a) directly in the content of PSM class when it is a structural representative and it has an element label or (b) when the structural representative does not have an element label, the content is propagated upwards to the closest significant node.

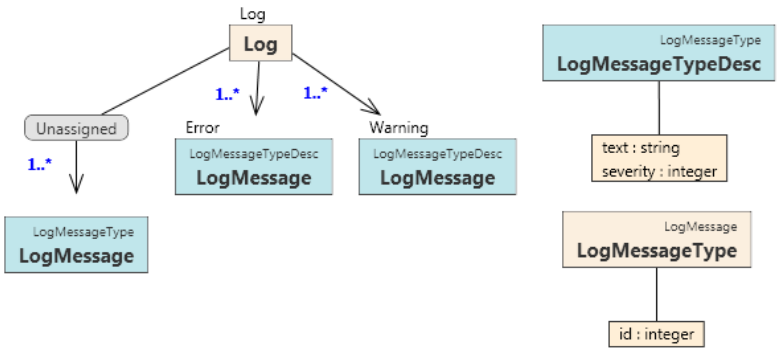


Figure 6.13: Structural Representatives – Propagation

Figure 6.13 contains a simple diagram with several structural representatives. The base type is `LogMessageType` with a single attribute `id`. This is referenced from structural representative classes `LogMessageTypeDesc` and `LogMessage` (in container `Unassgined`). `LogMessageTypeDesc` is referenced from classes with labels `Error` and `Warning`.

Classes with labels `Error` and `Warning` respectively fall the category (a) – the content inherited from `LogMessageTypeDesc` will be a part of the content of `Error` and `Warning` XML nodes. Class `LogMessage` on the left-hand side does not have an element label and thus the inherited content is propagated upward to the content of node `Unassigned`.

Appendix C.1 contains a sample document for the diagram in Figure 6.13.

Algorithm Gather Subelements (Algorithm 7) creates *structural representative elements* constructs for (a) (line 6) and (b) (line 25). Algorithm Gather Attributes is modified in a similar manner.

Algorithm 7 Gather Subelements (4)

```
1 function GetSubtreeElements(N' ∈ D'.𝒩)
2     returns subelements_N' – ordered sequence of node element wrapper s.
3 {
4     subelements_N' ← ∅
5     if N' ∈ C_psm ∧ N'.label ≠ null ∧ N'.Represented ≠ null
6         subtreeElements ← new structural representative elements(N',
7                 N'.Represented)
8
9     foreach node C' ∈ child nodes of N'
10        getSubtreeElementsInclRoot(N', var subelements_N')
11    return subelements_N'
12 }
13
14 procedure getSubtreeElementsInclRoot(N' ∈ D'.𝒩, var subelements_N')
15 {
16    if N' ∈ {C' ∈ C_psm|C'.label ≠ null} ∪ 𝒞𝒞 // simple element
17        subelements_N' ← new node element wrapper(N')
18    if N' ∈ 𝒜𝒞 // simple elements
19        foreach attribute a' ∈ N'.𝒜tt_psm
20            subelements_N' ← new node element wrapper(a')
21    if N ∈ D'_group // groups
22    {
23        var components
24        if N is structural representative
25            components ← new structural representative elements(N',
26                N'.Represented)
27        foreach node C' ∈ child nodes of N'
28            getSubtreeElementsInclRoot(C', var components)
29        subelements_N' ← new content group(N', components)
30    }
31    if N' ∈ 𝒞ℋ ∨ N' ∈ 𝒞𝒰 // choices, unions
32    {
33        // same as in Algorithm 6
34        ...
35    }
36 }
```

6.10.2 Templates for Structural Representative

The main purpose of structural representatives is to reuse parts of the model. This reusability should also be reflected in the revalidation script. If there are changes in the reused part of the model, it would be unsuitable to create the same revalidation instructions in each location where the particular part of the model is referenced.

To achieve reusability, templates for a red node N' that is referenced from structural representative(s) is slightly modified – its attributes and subelements are created in two separate templates $srAttTemplate_{N'}$ and $srElmTemplate_{N'}$ respectively. These templates are then called from the node itself and from all the structural representatives. Template 6.22 shows a templates for a red node referenced from structural representatives and Template 6.23 the definition of $srElmTemplate_{N'}$ ($srAttTemplate_{N'}$ is analogous).

```
<xsl:template ... >
    { if (N' ∈ D'_sign) }
        <{name(N')}> // wrapping tag for significant nodes
    { end if }
    { if N' ∈ D'_group }
            <xsl:call-template name='{srAttTemplate_N'}'>
                <xsl:with-param name='cg' select='$cg' />
            </xsl:call-template>
            <xsl:call-template name='{srElmTemplate_N'}' />
                <xsl:with-param name='cg' select='$cg' />
            </xsl:call-template>
    { else }
            <xsl:call-template name='{srAttTemplate_N'}' />
            <xsl:call-template name='{srElmTemplate_N'}' />
    { end if }
    { if (N' ∈ D'_sign) }
        </{name(N')}> // for significant nodes, close the tag
    { end if }
</xsl:template>
```

Template 6.22: Template for Nodes Referenced from Structural Representatives

```
<xsl:template name='{srElmTemplate_N'}' >
    { if N' ∈ D'_group }
        <xsl:parameter name='cg' />
    { end if }
    { process elements(GetSubtreeElements(N')) }
</xsl:template>
```

Template 6.23: Templates for Elements of a Referenced Node

Since we added new types of *node attribute wrapper/node element wrapper*, subroutines**process elements/attributes** must be again updated. The updated version of **process elements** is depicted in Template Fragment 6.24, subroutine **generate sr elements reference** (generating a reference of elements from structural representative) is depicted in Template Fragment 6.25. Subroutine **process attributes** is modified analogously and calls and **generate sr attributes reference** analogous to **generate sr elements reference**.

```
// process elements subroutine
process elements(elm : list of node element wrappers)
{
    foreach node element wrapper n_w ∈ elm
    {
        if n_w is simple node element
            generate element reference with
                context.CurrentElement = n_w
        else if n_w is content group
            generate group reference with
                context.CurrentContentGroup = n_w
        else if n_w is union elements
            generate choice elements reference with
                context.CurrentChoiceElements = n_w
        else if n_w is choice elements
            generate union elements reference with
                context.CurrentUnionElements = n_w
        else if n_w is structural representative elements
            generate sr elements reference with
                context.CurrentUnionElements = n_w
    }
}
```

Template Fragment 6.24: Templates for Elements and Attributes of a Referenced Node

```
{ if context.ForceCallable
    if forceCallableSrElmTemplate_N' not generated yet
        generate forceCallableSrElmTemplate_N' }
    <xsl:call-template name='{forceCallableSrElmTemplate_N'}' />
{ else }
    <xsl:call-template name='{srElmTemplate_N'}'>
        { if in group }
            <xsl:with-param name='cg' select='$cg' />
        { end if }
    </xsl:call-template>
{ end if }
```

Template Fragment 6.25: Generating SR Elements Reference

The process of creating force callable templates for attributes and elements – $forceCallableSrElmTemplate_{N'}$ and $forceCallableSrAttTemplate_{N'}$ is identical to creating force callable templates for nodes and groups.

The last step to complete the support of structural representatives is to extend values of `match` attributes of the top level templates (i.e. all templates of type *appliedNodeTemplate* and also of the two diagram templates $greenNodesTemplate_{D'}$ and $blueNodesTemplate_{D'}$).

The idea is that for each node M' that is in a subtree of a class referenced from a structural representative, the node models content at several locations in the document. The value of `match` attribute must be ready to accept all locations. Example in Figure 6.14 demonstrates this.

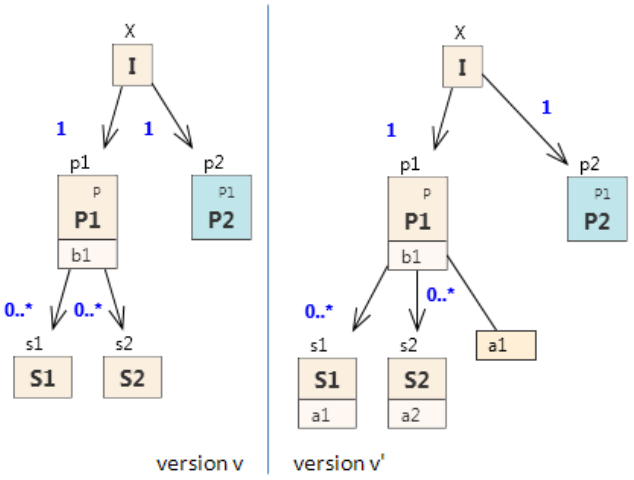


Figure 6.14: Match Attribute of Top Level Templates

```
<xsl:template match="/X/p1">
  <p1>
    <xsl:call-template name="X-p1-SR-ATT" />
    <xsl:call-template name="X-p1-SR-ELM" />
  </p1>
</xsl:template>

<xsl:template name="X-p1-SR-ATT">
  <xsl:attribute name="b1" />
</xsl:template>
<xsl:template name="X-p1-SR-ELM">
  <xsl:apply-templates select="s1" />
  <xsl:apply-templates select="s2" />
  <xsl:call-template name="X-p1-a1" />
</xsl:template>

<xsl:template match="/X/p2/s1 | /X/p1/s1">
  <s1>
    <xsl:attribute name="a1" />
  </s1>
</xsl:template>
```

```
<xsl:template match="/X/p2/s2 | /X/p1/s2">
  <s2>
    <xsl:attribute name="a2" />
  </s2>
</xsl:template>
<xsl:template name="X-p1-a1">
  <a1></a1>
</xsl:template>

<xsl:template match="/X/p2">
  <p2>
    <xsl:call-template name="X-p1-SR-ATT" />
    <xsl:call-template name="X-p1-SR-ELM" />
  </p2>
</xsl:template>

<!--blue nodes template-->
<xsl:template match="/X">
   // usual blue node template body
</xsl:template>
```

Figure 6.15: Example Document for Diagram in Figure 6.14

The revalidation script in Figure 6.15 shows how `match` attribute was extended for nodes `s1` and `s2` in order to match the instances not only under node `p1` but also under the structural representative node `p2`. The template for node `p2` shows how templates ($srAttTemplate_{p1}$ and $srElmTemplate_{p1}$) are called from a structural representative node.

Chapter 7

Implementation and Experiments

7.1 Implementations

The prototype implementation of the XSem model – XCase was presented in [18].

- XCase is a full-fledged CASE tool for creating PIM and PSM diagrams using the XSem model.
- It implements the basic translation algorithm proposed in [24] for translation from an XSem-H PSM diagram into an XSD.
- It contains an experimental implementation proposed in [19] for importing an existing SD into an empty PSM diagram and creating a PIM diagram for this PSM diagram.
- It contains a simple generator of random XML documents valid against a PSM diagram.

As a part of this thesis, XCase was extended with following features:

- The ability to maintain several versions of the model.
- A *branch* function that duplicates a selected version of a model and connects the source and branched version of the model with *version links* (fills the relation $\mathcal{VL}$).

- Basic means of editing the relation $\mathcal{VL}$.
- A prototype implementation of the algorithm proposed in Chapter 6.

The prototype implementation generates a revalidation script following the revalidation algorithm (with several exceptions – changes of attribute types are ignored in the current implementation, as well as the changes of the *any attribute* flag and conversions from a regular PSM class into a structural representative and vice versa).

The revalidation script can be tested in XCase on random data (Saxon XSLT processor [6] is used for XSLT processing).

7.2 Experiments

Since there are no real-world projects that use the multi-version extension of XCase, experiments can be conducted only on the existing specifications in XML Schema language thanks to the possibility to import an XSD into a PSM diagram.

By importing two different versions of the same XSD and connecting the created constructs via version links, our approach can be used for generating evolution scripts even for the existing specifications.

RSS specification A smaller example is the specification of RSS, versions 0.90 and 2.0 [4] [5]. Although both versions model a similar XML structure, the structure of the PSM diagrams for the respective versions is different. The algorithm generates a correct revalidation script.

OpenTravel specification To test the approach on a larger example, two versions of the schema `FS_OTA_CancelRQ.xsd` that is a part of the OpenTravel Specification [1] (namely the versions 2008A and 2009A) were imported into the program using the XSD to PSM feature.

Most of the version links were created using the trivial matching algorithm that is able to connect unchanged constructs. Version links for the changed constructs were added manually.

Among open specifications, it is a common practice to create the new versions of the specifications backwards-compatible and the OpenTravel specification is also the case (OpenTravel schemas provide a large amount of freedom to the user in general, for example by declaring all the elements and attributes optional).

Nonetheless, the nature of the changes is too complex for the algorithm to recognize the backward-compatibility, so a non-empty revalidation script is created. However, the revalidation script is correct and outputs a valid document.

Chapter 8

Conclusion

The aim of the thesis was to propose an approach to XML schema evolution built upon a conceptual model for XML schemas. It should identify changes in the schema, determine the impact of the changes on the existing documents valid against the old version and produce a revalidation script when revalidation of the existing documents is necessary.

We began by naming the scenarios in which the existing systems using XML schemas evolve and what are the consequences of the evolution process (see Chapter 1). A survey of the existing approaches to the XML schema evolution followed in Chapter 2. In Chapter 3 we formally defined a conceptual model XSem for platform-independent and platform-specific modeling.

In Chapter 4 we extended the XSem conceptual model with the ability to model multiple versions of the model at once. Using version links, each construct in the model is correctly connected with its counterpart constructs in all other versions, where the construct exists.

Adding the version links allowed us to define changes (Chapter 5) that can occur between two versions v and v' of a diagram $\mathcal{D}$ and an algorithm that detects these changes. With each change we examined its impact on the validity of the documents valid against version v ($\mathcal{S}(\mathcal{D})$) and identified the set of changes that do not violate validity of documents in $\mathcal{S}(\mathcal{D})$.

A set of changes $C_{\mathcal{D},\mathcal{D}',v,v'}$ between versions v and v' of a diagram $\mathcal{D}$, which is the output of Algorithm 2, does not expect any particular implementation language for revalidation.

In Chapter 6 we proposed an algorithm that produces an XSLT script that revalidates a valid document $D \in \mathcal{S}(\mathcal{D})$ and produces a new document document $D' \in \mathcal{S}(\mathcal{D}')$ valid for version v'. The revalidated document preservers semantical

meaning of the constructs thanks to utilizing the version links defined in Chapter 4.

The main contribution of our approach is the ability to maintain an arbitrary number of different versions of each diagram with the ability to quickly generate the revalidation script for any two different versions. The revalidation script translates a document valid against the first version into a document valid against the second version.

In contrast to the "recording" approaches (where only the new version can be edited during the evolution process), the different versions can be edited separately. The editing can also be suspended and resumed any time, separate branches of any version can be created on demand and unlimited amount of versions can be maintained in the model.

Thanks to version links, the algorithm is able to correctly distinguish moving operations from adding/deleting, correctly handle renaming, reordering and complex composite changes in the structure of the document. The version links solve several issues inherent in algorithms discussed in Chapter 2.

The algorithm takes into account a large number of possible constructs at the platform-specific layer and thus covers a wide set of modeled XML schemas. Another advantage of the algorithm is that it can be applied regardless of the particular XML schema language. The XSem-H platform-specific diagram models a set of XML documents, it is not a visualization of an XML schema in any particular XML schema language. The revalidation algorithm examines the versioned XSem-H PSM diagrams, not the schemas themselves, therefore the independence of the XML schema language is preserved.

Integration with the XSem model with its PIM and PSM levels leaves an open field for further enhancements of the system. The PIM level can be further utilized for generating content (see Section 8.1.1 for details), new features can be added to the PSM constructs to further tweak the evolution process.

An experimental implementation of the algorithm was integrated into XCase and is available on the attached CD.

8.1 Future Work

Our approach takes into account almost all constructs defined in [24] and usefully extends the XSem model. To further extend its practical applicability, the future research can examine the following areas.

8.1.1 Generating Content

The revalidation script generated by the revalidation algorithm proposed in Chapter 6 is able to supply new content when needed. This content is always a default instance of some part of the XSem-H diagram (see Section 5.4).

Adding the default instance into the document makes the document valid, but does not contain any semantic value. However, the approach could be extended to add semantically correct content into the new document. For this purpose, PIM links could be utilized. A possible solution would be to generate the content based on the data retrieved from the database. The database query could be also automatically generated, if the database structure would correspond to the PIM model.

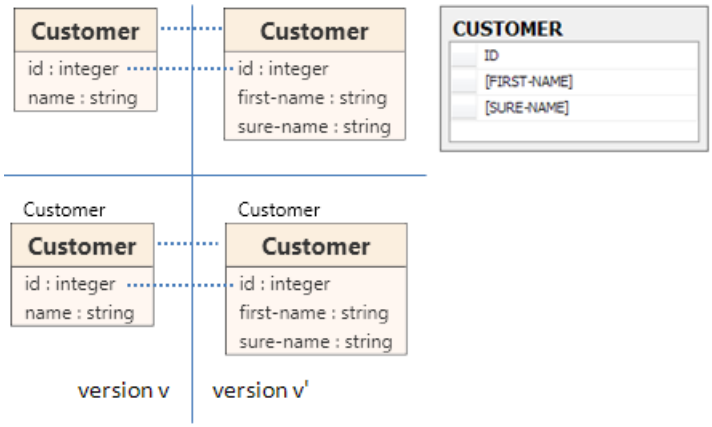


Figure 8.1: Adding Content

Consider the model in Figure 8.1. It contains a PIM class `Customer` in version v and v' and PSM class derived from it in a PSM diagram. The attribute `name` was removed and replaced by attributes `first-name` and `sure-name` in version v'. The revalidation script generated by the current algorithm would be:

```
<xsl:template match="/Customer">
 <Customer>
   <xsl:copy-of select="@id" />
   <xsl:attribute name="first-name" />
   <xsl:attribute name="sure-name" />
  </Customer>
</xsl:template>
```

If we extended the algorithm with considering PIM linked to a relational database, it could produce a revalidation script like[1]:

```
<xsl:variable name="first-name">
    {select [FIRST-NAME] from CUSTOMER where ID="/Customer/@id"}
</xsl:variable>
<xsl:variable name="last-name">
    {select [FIRST-NAME] from CUSTOMER where ID="/Customer/@id"}
</xsl:variable>
<xsl:template match="/Customer">
 <Customer>
   <xsl:copy-of select="@id" />
   <xsl:attribute name="first-name">
     <xsl:value of select="$first-name" />
   </xsl:attribute>
   <xsl:attribute name="sure-name">
    <xsl:value of select="$last-name" />
   </xsl:attribute>
  </Customer>
</xsl:template>
```

The most appropriate solution would be to add another type of platform-specific model – a model of a relational database. This model would be linked to the PIM model in a similar way as XSem-H diagrams are and the required queries could be inferred from the links between XSem-H and PIM and PIM and the relational database model.

8.2 Version Links for Imported Schemas

XCase contains an experimental implementation of an algorithm proposed in [19] for importing an existing XSD into an XSem-H diagram. Combined with the algorithm proposed in Chapter 4, the system can be used to generate revalidation scripts in those scenarios, where we already have both old and new version of the XSD. The old schema is imported into diagram $\mathcal{D}$, new schema into $\mathcal{D}'$.

The only missing part for the successful run of the algorithm are the contents of relation $\mathcal{VL}$ (version links joining the previous and the evolved version of each construct). This relation can be created manually by the user, but it is a time-consuming process.

[1] We use a symbolic notation for incorporating SQL queries into XSLT script. The correct notation is vendor-specific.

It would be useful to extend the system with a heuristic that could create a larger part of $\mathcal{VL}$ automatically and ask for user input only in the unresolved cases. A trivial matching algorithm is already a part of XCase, but now it can only connect constructs in unchanged parts of the diagram.

There exists a lot of methods for finding similarities and patterns among two XML schemas (a survey can be found in [25]). Outcomes of these methods can be transferred into finding similarities between two XSem-H diagrams.

8.3 Generalizations and Extensions

XSem model defined in Chapter 3 lacks support for inheritance. However, translation of inheritance at the level of platform-specific XSem-H diagrams to XML Schema constructs was proposed in [24]. Adding support for inheritance into revalidation involves two tasks:

- handling generalization already present in the diagram in both versions v and v' (changes occurred either in the general or in the specific class)
- handling generalizations added/removed in version v'.

Appendix A

CD Contents

The attached CD contains:

- PDF version of this thesis - *thesis.pdf*
- XCase installer (includes a prototype implementation of the algorithm described in Chapter 6)
- Examples for diagram evolution
- Definitions of the diagram-independent auxiliary templates - *auxiliary-templates.xslt*

Running the Examples Examples are stored in the folder `Evolution examples`. Each example is an `.XCase` project file. When the file is opened in XCase, select the new version of the diagram and click *Find changes* in the main toolbar[1] to start the change detection algorithm and display the changes. To generate the revalidation stylesheet, click *Evolve* and then *XSLT from changes*.

The generated revalidation stylesheet can be tested in the same window on some random input files (generated after clicking *Another sample*).

Examples 1-5 are described in this thesis (Figures 6.1, 6.3, 6.9, 6.11 and 6.14).

Example 6 contains two versions of a schema for RSS - version 0.90 [4] and 2.0 [5]. Both versions were obtained first by translating the DTD into the XSD (via Visual Studio 2008) and then importing the XSD into XCase.

Example 7 contains two versions of schema `FS_OTA_CancelRQ.xsd` that is a part of the OpenTravel Specification [1] (namely the versions 2008A and 2009A).

[1] On machines with smaller display resolution, the button may be hidden and accessible after clicking the arrow on the right of the toolbar.

Appendix B

Sample XML Document and XML Schema Translation for Diagram 3.14

XML Schema Translation and a valid document for diagram from Figure 3.14 (p. 47).

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="catalog" type="CategoryType"/>

  <xs:complexType name="BookType">
    <xs:sequence>
      <xs:element name="author" type="xs:string"/>
      <xs:element name="title" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="CategoryType">
    <xs:sequence>
      <xs:element name="description" type="xs:string" />
      <xs:element name="category" type="CategoryType"
        minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="books">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="book" type="BookType"
              minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
```

```
      </xs:element>
      <xs:element name="best-seller" type="BookType"
        minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="code" type="xs:string" />
    <xs:attribute name="title" type="xs:string" />
  </xs:complexType>

</xs:schema>

<?xml version="1.0"?>
<root-category code="1" title="Main">
  <category code="1.1" title="Fiction">
    <description>All fiction books</description>
    <category code="1.1.1" title="History">
      <description>History books</description>
      <books>
        <book>
          <title>#</title>
          <author>#</author>
        </book>
        ...
      </books>
      <best-seller>
        <author>#</author>
        <title>#</title>
      </best-seller>
    </category>
    <category code="1.1.2" title="Drama">
      <description>Drama books</description>
      ....
    </category>
    ...
    <books />
  </category>
  <category code="1.2" title="Non-fiction">
    <description>All non-fiction books</description>
    ...
  </category>
  <books />
</root-category>
```

Appendix C

Sample XML Document for Diagram 6.13

```
<Log>
  <Unassigned>
    <id>1</id>
    <id>2</id>
  </Unassigned>
  <Error>
    <id>1</id>
    <text>Missing ';'.</text>
    <severity>1</severity>
  </Error>alg:change-detection
  ...
  <Error>
    <id>2</id>
    <text>Method named 'Create' already exists</text>
    <severity>1</severity>
  </Error>
  <Warning>
    <id>3</id>
    <text>Method Show() is obsolete.</text>
    <severity>5</severity>
  </Warning>
</Log>
```

Figure C.1: Sample Document for Diagram from Figure 6.13

Bibliography

[1] *OpenTravel Specification*. http://www.opentravel.org/.

[2] Oracle XML DB Developer's Guide - Oracle XML Schema Annotations. http://download-uk.oracle.com/docs/cd/B28359_01/appdev.111/b28369/xdb05sto.htm#i1030452.

[3] Oracle XML DB Developer's Guide - XML Schema Evolution. http://download-uk.oracle.com/docs/cd/B28359_01/appdev.111/b28369/xdb07evo.htm#BCGFEEBB.

[4] *RSS 0.90 Specification*. http://www.rssboard.org/rss-0-9-0.

[5] *RSS 2.0 Specification*. http://cyber.law.harvard.edu/rss/rss.html.

[6] *Saxon XSLT Processor*. http://saxon.sourceforge.net/.

[7] XCase - tool for XML data modeling. http://xcase.codeplex.com/.

[8] A. Khan, M. Sum. Introducing Design Patterns in XML Schemas. http://developers.sun.com/jsenterprise/archive/nb_enterprise_pack/reference/techart/design_patterns.html.

[9] D. Booth, C. K. Liu. *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*. W3C, June 2007. http://www.w3.org/TR/wsdl20-primer/.

[10] E. Domínguez, J. Lloret, A. L. Rubio, and M. A. Zapata. Evolving xml schemas and documents using uml class diagrams. In K. V. Andersen, J. K. Debenham, and R. Wagner, editors, *DEXA*, volume 3588 of *Lecture Notes in Computer Science*, pages 343–352. Springer, 2005.

[11] G. Guerrini, M. Mesiti. XML Schema Evolution and Versioning: Current Approaches and Future Trends.

[12] G. Guerrini, M. Mesiti, and M. A. Sorrenti. Xml schema evolution: Incremental validation and efficient document adaptation. In D. Barbosa, A. Bonifati, Z. Bellahsene, E. Hunt, and R. Unland, editors, *XSym*, volume 4704 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2007.

[13] Hong Su, D. K. Kramer, E. A. Rundensteiner. XEM: XML Evolution Management.

[14] ISO. *ISO/IEC 9075-14:2008 - SQL – Part 14: XML-Related Specifications (SQL/XML)*. http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=45499.

[15] M. Kay. *XSLT 2.0 and XPath 2.0 4th Edition*. Wrox, 2008.

[16] M. Kay. *XSL Transformations (XSLT) Version 2.0*. W3C, January 2007. `http://www.w3.org/TR/xslt20/`.

[17] 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, *Workshop Proceedings Datenbanksysteme in Business, Technologie und Web (BTW 2007)*, pages 53–63, Aachen, Germany, March 2007.

[18] 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.

[19] J. Klímek. Xml Schema Evolution, 2009.

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

[21] I. Mlynkova and J. Pokorny. Five-level multi-application schema evolution. 2009.

[22] M. M. Moro, S. Malaika, and L. Lim. Preserving xml queries during schema evolution. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 1341–1342, New York, NY, USA, 2007. ACM.

[23] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Internet Technol.*, 5(4):660–704, 2005.

[24] M. Necasky. *Conceptual Modeling for XML*, volume 99 of *Dissertations in Database and Information Systems Series*. IOS Press/AKA Verlag, January 2009.

[25] M. Nečaský and I. Mlýnková. Exploitation of similarity and pattern matching in xml technologies. In *DATESO 2009*, volume 471 of *CEUR Workshop Proceedings*, pages 90–104. MatfyzPress, 2009.

[26] Object Management Group. *UML Infrastructure Specification 2.1.2*, nov 2007. http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/.

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

[28] A. M. P. V. Biron, K. Permanente. *XML Schema Part 2: Datatypes (Second Edition)*. W3C, October 2004. `http://www.w3.org/TR/xmlschema-2/`.

[29] S. Systems. Enterprise Architect. `http://www.sparxsystems.com.au/products/ea/`.

[30] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. W3C, September 2006. `http://www.w3.org/TR/REC-xml/`.

[31] B. Thalheim. *Entity-Relationship Modeling: Foundations of Database Technology*. Springer Verlag, Berlin, Germany, 2000.

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

[33] C. M. S.-M. Tim Bray, Jean Paoli. *Document type declaration*. 2000.

[34] W3C. *Document Object Model (DOM) specification*. http://www.w3.org/DOM/.

[35] W3C. *XML Path Language (XPath) 2.0*. http://www.w3.org/TR/xpath20/.

[36] W3C. *XQuery 1.0: An XML Query Language*. http://www.w3.org/TR/xquery/.

[37] W3C. *XQuery Update Facility 1.0 specification*. http://www.w3.org/TR/xquery-update-10/.

[38] W3C. *XQuery Update Facility 1.0 specification*. http://www.w3.org/TR/xquery-update-10/.