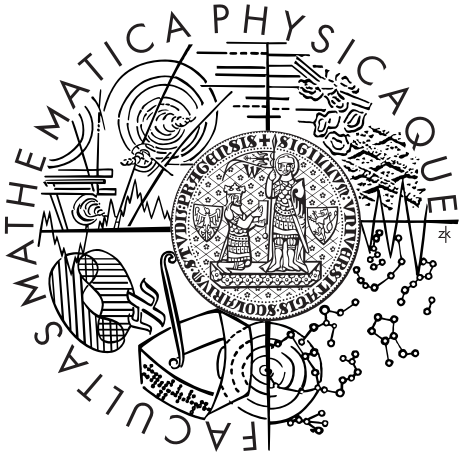
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Jakub Malý

XML Document Adaptation and Integrity Constraints in XML

Department of Software Engineering

Supervisor of the doctoral thesis: Mgr. Martin Nečaský, Ph.D.

Study programme: Computer Science

Specialization: Software Systems

Prague 2013

I would like to thank my supervisor, Mgr. Martin Nečaský, Ph.D., for his counsels during my doctoral studies. I would also like to thank RNDr. Irena Holubová, Ph.D. for her advice and support. Finally, I would like to thank Michael Howard Kay, Ph.D FBCS, for the possibility of using Saxon EE to verify my results.

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Adaptace XML dokumentů a integritní omezení v XML

Autor: Jakub Malý

Katedra: Katedra softwarového inženýrství

Vedoucí disertační práce: Mgr. Martin Nečaský, Ph.D., Katedra softwarového inženýrství

Abstrakt: Práce se zabývá managementem a konzistencí XML dat – přesněji problémem adaptace dokumentů a užití integritních omezení. Změny požadavků uživatelů často způsobují změny ve schématech používaných v systému a následně způsobí, že stávající dokumenty se stanou nevalidními. V práci představíme formální framework pro detekci změn mezi dvěma verzemi schématu a generování transformace z původní na novou verzi schématu. Rozsáhlé informační systémy spoléhají na platnost integritních omezení. V práci ukážeme, jak lze pro definici omezení v XML datech na abstraktní úrovni použít jazyk OCL a jak lze tato omezení přeložit do výrazů jazyka XPath, schemat jazyka Schematron a ověřit tak jejich platnost.

Klíčová slova: XML schema, evoluce schémat, adaptace dokumentů, integritní omezení

Title: XML Document Adaptation and Integrity Constraints in XML

Author: Jakub Malý

Department: Department of Software Engineering

Supervisor: Mgr. Martin Nečaský, Ph.D., Department of Software Engineering

Abstract: This work examines XML data management and consistency – more precisely the problem of document adaptation and the usage of integrity constraints. Changes in user requirements cause changes in schemas used in the systems and changes in the schemas subsequently make existing documents invalid. In this thesis, we introduce a formal framework for detecting changes between two versions of a schema and generating a transformation from the source to the target schema. Large-scale information systems depend on integrity constraints to be preserved and valid. In this work, we show how OCL can be used for XML data to define constraints at the abstract level, how such constraints can be translated to XPath expressions and Schematron schemas automatically and verified in XML documents.

Keywords: XML schema, schema evolution, document adaptation, integerity constraints

Contents

Introduction

The eXtensible Markup Language (XML) [9] is a standard widely used for data representation and interchange, gaining popularity as native storage format and, together with the accompanying languages and tools (the so called XML stack), such as XML Schema Definition (XSD) [92], XPath [89], XQuery [6], XSLT [87], XProc [90] etc., it becomes a very powerful technology.

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

When the amount and overall complexity of the used interrelated XML formats exceeds certain level, it is useful to create a conceptual model of the application, which describes the used real-word entities without the clutter and excessive verbosity often associated with XML technologies. A software engineering standard – Unified Modeling Language (UML, [23]) – can be used in such cases ([25], [65]).

Complex systems also often rely on certain invariants and data-integrity constraints (ICs) that are required for the system to function properly. However, these constraints often can not be properly described by the visual languages of UML diagrams. In the cases where diagrams are not expressive enough, Object Constraint Language (OCL, [66]) can be used to describe additional properties and conditions of the system. OCL is a formal language and thus its expressions defined at the abstract layer can be used by the transformations and propagated to the specific layers. A system using a family of XML formats often uses some of the XML formats to define its interface to the outside world, e.g., using the technologies of Web Services [85] and languages WSDL [86] and XML Schema. Documents on the inputs are checked against schemas, usually one of the grammar-based (XML Schema, Relax NG [12], DTD [82]), sometimes also using other validation technologies (Schematron [29], NVDL [30]). Particularly Schematron is a suitable technology to verify integrity constraints in the contents of the document.

Problem definition Having a system which exploits a family of XML formats, we face to the problem of *XML format management and evolution* as a specific part of management and evolution of the software system as a whole. The XML formats may need to be evolved whenever user requirements or surrounding environment changes. In our previous work [61, 56], we have introduced a framework for such modeling, management and evolution of a family of XML formats. The

framework considers different levels of abstraction of the XML formats, e.g., conceptual schemas, logical XML schemas or instance XML documents. These levels are mutually interrelated by mappings to assure consistency among different XML formats in the family. In our framework, the user defines the representations of the real world concepts used in the system (and their relationships) at the abstract level and references (reuses) the common definitions from the levels below. In [56] we solved a problem of coherent management and evolution of XML formats according to changing requirements (*schema evolution*). This means that when a new requirement appears, it is implemented in the XML formats so that they are updated coherently with each other.

In this thesis, we address two areas related to XML format management. First, the area of integrity constraints and modeling and management of schemas that verify integrity constraints and data present in the document. We show that integrity constraints can be managed in a similar manner as the definitions of real-world concepts and their relationships. The integrity constraints (in the form of expressions) are defined at the abstract level and can be reused by the levels below automatically. One constraint can be relevant for several XML formats and thus should be checked in by a schema for each of the particular format.

Second, we address the problem tightly related to *schema evolution* is *document adaptation*. To adapt documents after schema evolution means to transform documents valid against the old version of the schema into documents valid against the new version of the schema. It can also be looked upon as propagation of changes from the schema (after schema evolution) to the documents in the system. This problem is also related to integrity constraints, because the changes in the schema might be not only changes in structure, but also changes in the semantics of the concepts. The semantics can be described using integrity constraints and document adaptation algorithm should take them into account.

Contributions In this thesis we focus on the impact of evolution of schemas of XML formats on their instance XML documents. We extend the framework previously published by XML and Web Engineering Research Group ([56]) with versioning features and an algorithm for propagation of changes to XML documents – i.e., an algorithm for *XML document adaptation*. We also extend the framework with support for the standard Object Constraint Language to create expressions, especially integrity constraints and invariants. We utilize OCL expression to let the user to define annotations to his model and thus accurately describe the semantics of adaptation scenarios. Our main contributions can be summed up as follows:

- We propose an extension of our previously published framework for evolution of XML formats with versioning features. The XML schemas of XML formats are edited by designers at a more user-friendly level of conceptual schemas instead of technical XML schemas.
- We provide an algorithm for automatic detection of changes between any two versions of an XML schema and propagation of changes to instance XML documents (in a form of automatically generated version transformation script).
- The framework can decide automatically whether transition from one ver-

sion of the schema to another requires transformation of the existing documents to make them valid against the new version.

- We add support for both defining and verifying integrity constraints in XML formats.
- The adaptation algorithm can be enhanced with OCL annotations, which ensure correct adaptation in the ambiguous cases.
- We allow the user to declare complex relationships between the old and new version of the schema using a formal language at the conceptual level.
- In the thesis, we provide a detailed description of the algorithms and the theoretical background. However, a proof-of-concept implementation was also developed as a part of this research and is publicly available [43].

Outline The rest of the thesis is structured as follows: In Chpt. 1, we present motivational scenarios for integrity constraints and document adaptation and general requirements for an adaptation framework. In Chpt. 2, we introduce our five-level framework and describe the purposes and responsibilities of each level. The conceptual model of the platform-independent and platform-specific level is introduced formally in Chpt. 3 followed by formal model for versioning. Chpt. 4 describes the adaptation framework and algorithms. Chpt. 5 introduces OCL as an expression language for the platform-independent and platform-specific level, proposes required extensions of the standard OCL and describes the algorithm for translating OCL expressions into XPath expressions. Chpt. 6 combines the results presented in Chpt. 5 and Chpt. 6 and enhances the adaptation algorithm with OCL annotations. In Chpt. 7, we describe the experimental implementation. In Chpt. 9, we outline the open problems, room for improvement and course of our future research. In Chpt. 9.4 we conclude.

1. Motivation and Requirements

In this chapter, we discuss motivation for the extensions of our framework with integrity constraints and document adaptation.

1.1 Integrity Constraints

The idea behind Model Driven Development (MDD) [52] is to model the software system on several layers varying in their degree of abstraction. A designer starts from the very abstract specification (independent of the platform and language used) and progresses to more concrete models (using platform-specific constructs) and finally to code. Ideally, each step of the transformation of the model from the more abstract to the less abstract is achieved by a declarative transformation obtained (semi-)automatically. In a system using a family of XML formats, using this approach ensures to preserve consistency across all the formats, because the concepts (e.g., `Employee`, `Department`) and their relations are defined at an abstract level (in one place), which prevents inconsistent usages or redefinitions of the same concept in two (or more) XML formats.

If we add integrity constraints into the system, we would like to achieve a similar level of abstraction and consistency. An integrity constraint defined at the abstract level is valid for the whole problem domain, independently of the XML formats and schemas existing in the system. However, when the system uses multiple XML formats, the data are often represented in different structures for each document type according to its XML schema. Therefore, for validation purposes, it is necessary to express the constraint as a validation rule for each relevant XML format in a suitable XML validation language, e.g., Schematron and each translation of the integrity constraint into an expression will be meaningful only for the particular format.

It is ineffective to express the constraint directly in a validation language manually. There can exist various relevant XML schemas for each constraint but with a different structure. Therefore, a designer has to express the constraint specifically for each separate XML schema in a form of different validation rules. This can be very time-consuming and also error-prone. Also, when the system is evolved and schemas are changed, all the validation expression must be examined and adapted to the new version.

We propose a more effective method. We start by expressing each domain integrity constraint only once – as a part of the abstract level. This is practical because the designer can define the integrity constraints in the phase of the domain analysis, before creating particular XML formats. Later, when XML formats are derived from the abstract level, each integrity constraint is examined from the point of view of every XML format in the system. The algorithm decides, whether the particular constraint is relevant for the particular XML format and if so, the constraint is translated into an expression that can be validated in the particular XML format (documents valid against the XML schema of that format) using a standard language from the XML stack.

1.2 Adaptation Scenarios

In this section, we provide three scenarios where a document adaptation approach might be applied.

Document update after new version adoption In the first scenario, we consider an XML application storing data in XML documents. The documents can be stored in a file system, in an XML-enabled relational database shredded into tables [68] or in a native XML database [19, 4, 49]. As requirements change, the system designer needs to adjust the XML schemas existing in the system. To keep the system consistent, the documents stored in the system must be transformed so that they are valid against the new version of the schemas. The process of propagation of changes from schemas to documents is called *document adaptation*.

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

In case where XML documents are stored as files or in a native database, the adaptation script can be executed upon them directly. When the documents are stored in a relational database, the database vendor usually provides an interface [67] using which transition to the new version is performed – this interface requires the evolved schema and the adaptation script. Using our approach, the user only needs to evolve the schema, the adaptation script is generated for him/her.

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

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

results back to the version $\tilde{v}$ when sending them as a response).

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

Our framework significantly reduces the effort required when the first way of dealing with different versions of documents on the input/output is selected. It can generate an adaptation script for any two versions of the schema and this adaptation script can be used to pre-process the documents sent by the parties that have not adopted the latest versions yet in the mediator (and the process is transparent for the business logic components). The only requirement that the designer needs to evolve the old version to the new version in our framework which keeps track between both versions.

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

In [34], we describe, how our framework deals with the integration problem and helps the user to define mappings between systems. The document adaptation algorithm can utilize these mappings to create an adaptation script and again supply the systems with mediator components (and use generated adaptation scripts in these), so that they will be able to communicate with the other systems using different set of schemas.

1.3 Evolution/Adaptation Framework Requirements

In this section, we will elaborate on the universal requirements for an evolution/adaptation framework. We formulate them generally, without presuming a specific implementation language or adaptation workflow.

Rich set of supported operations The power of an evolution framework is to a large degree determined to the size of the set of supported operations (sometimes called evolution primitives). The absolute minimum are the operations that *create* and *remove* the individual parts of the XML schema. The actual amount of different operations depends on the level of abstraction and the XML schema

language used. E.g., when the framework works directly with DTDs, it will require operations such as *create/remove element/attribute definition*. A framework working with XSDs will require a wider set of operations, because XSDs provide a wider set of constructs. A framework that abstracts the schema using UML will require operations such as *create/remove class/attribute/association*.

Supporting only *create/remove* kind of operations can be insufficient for some kinds of adaptation frameworks – those that propagate the schema editing operations directly to the adapted XML documents. In that case, every *modification* is actually a pair of *create* and *remove* and when this pair of operations is propagated, the data is lost, not modified. Some frameworks support *modification* scenarios by adding additional operations: *move* operations and *modify property* operations (e.g., modify the `minOccurs` property of an `xs:sequence` in an XSD).

It must be noted that not even adding *modify property* P operation for every existing property P in the language or model used by the system can cover all the adaptation scenarios, if the system implements document adaptation as propagation of individual schema evolution operations. In general, each time an adaptation scenario concerns multiple constructs and properties and it must be broken down into several operations, it may not be adapted properly. Examples of such scenarios are splitting an XML attribute into two new attributes (by a separator character) or adding a element attribute whose value should be computed from other attributes etc. Splitting an attribute A into attributes B and C may be implemented e.g., as a sequence:

1. *remove attribute* A
2. *add attribute* B
3. *add attribute* C

alternatively:

1. *modify attribute* A *into* B (this operation may have to be broken into several operations)
2. *add attribute* C.

When the first operation from the first sequence is propagated, the value of A is lost and the system will not be able to determine the value of B and C. When the first operation from the second sequence is propagated, the value of B will be the formal value of A and again, the system will not be able to determine the value of C. There may be other translations, but when the system only works at the schema level, the adaptation will not be correct unless the whole scenario is recognized as one operation (more on this in the requirement Semantics of operations further in this section).

The system might add additional operations for the more frequent scenarios (e.g., a *split attribute* operation), but the number of supported operations is always finite.

Separation of evolution of schemas and adaptation of documents It is highly useful when schema evolution and document adaptation are separated phases of system evolution. The main advantage is that it allows to perform more operations in one cycle. The main reason is that the schema evolution is a process

that should be interactive and responsive (e.g. the user may use a CASE[1] tool to edit the schema) and it should not be influence by the amount of documents in the schema.

On the contrary, document adaptation should not be interactive in the ideal case (once the adaptation script is created, it can be applied to any document valid against the old version of the schema). The reason for this is that the number of documents requiring adaptation can be very high and thus every adaptation cycle might take a large amount of time. Approaches that interleave schema evolution and document adaptation (allow only one schema evolution operation in each adaptation cycle) thus have a certain limit in the amount of documents they can support (equal to the number of documents the system can process in a time the user is willing to wait after each edit operation).

Finally, when the framework is designed in such a way that it has to propagate the changes immediately, it may be used only in those scenarios where all the documents that must be adapted are present in the system – from the scenarios presented in Sec. 1.2 at the beginning of this chapter, only the first one satisfies this condition.

Normalization before propagation If more than one operation in each evolution cycle is allowed, the sequence of operations should be normalized to eliminate repetitive and redundant operations (e.g., when the user edits the same property of a construct repeatedly or performs several operations that cancel each other). A *normalized sequence* of operations has exactly the same outcome as the original sequence, but is shorter (simpler, more efficient). The phase of normalization is not required for the correctness of the framework, but can improve its performance significantly, because the benefits of normalization are increasing with the size of the set of adapted documents.

Semantics of operations, references to content of documents We have already observed some problems which emerge when the adaptation framework operates only at the schema level and does not allow the user to refer to the contents of the documents when discussing the required set of operations. In general, some adaptation scenarios depend on the contents of document and the adaptation script should behave differently for different documents based on the values and contents of the particular document.

Sometimes, it is also necessary to perform adaptation of documents even when the schema does not change from the grammatical point of view (and a grammar-based XML schema definition, such as a DTD or XSD, would be the same in both the original and the evolved version). Again, it is necessary that the framework allows the user to define not only structure, but also the relations between the content of the source and adapted documents.

Resolving ambiguities In some cases, when a schema is evolved, there are multiple options how to adapt the documents to the change. The user should be at least notified by the system that some operation may be interpreted in multiple ways. A more advanced solution is either allow the user to explicitly

[1] Computer Aided Software Engineering

declare the exact semantics of the operation (by providing some additional information) or letting him choose the desired adaptation where it cannot be decided automatically.

Advanced content generation When definitions of new attributes, elements or whole subtrees are added to the schema, the attributes, elements and/or subtrees must be created in the adapted documents (if they are not added as optional). When the respective data cannot be obtained from the source document itself, they must be generated. Prevailing approach to this issue is either to create some kind of minimal/default content (empty subtrees, attributes with default values etc.) or to let the user specify the content manually. However, the system can have the required content stored already, only in some other data source (e.g., in a relational database or other XML document). Framework designed upon a conceptual model can exploit this – e.g., by automatically generating a query (in SQL, SQL/XML, XQuery) to retrieve the required data for the revalidated document.

Other aspects Finally, there are other aspects, that should be taken into consideration, such as whether the framework has a *formal background* and whether it provides some form of *graphical notation*. The level of abstraction the framework provides is also an important criterion, which is, in fact, not universal. Some user's may prefer a more high-level approach (which abstracts from the technical terms of the underlying technology), another may prefer a tool that uses the terminology and constructs of the implementation language (e.g., *global* and *local types*, *substitution groups* etc. in the case of XSDs), because it may give him a finer control over the results.

2. 5-Level Framework for Design and Evolution of XML Formats

Our five-level evolution framework was partly introduced in [54, 60, 58, 59], its updated version in [56]. The framework is a joint outcome of the collaboration within XML and Web Engineering Research Group. In this Chapter, we introduce the framework as a whole, in Section 2.2, we specify which parts of the framework play a role in this thesis.

The schema of the framework is depicted in Figure 2.1. It is partitioned both horizontally and vertically. Vertical partitions represent individual XML formats. Horizontal partitions represent different levels which characterize each of the XML formats from different viewpoints:

- The *extensional level* contains XML documents formatted according to the XML format.
- The *operational level* contains operations performed on the XML documents from the extensional level. These can be queries over the instances or transformations of the instances.
- The *logical level* contains a logical XML schema which specifies the syntax of the XML format. It is expressed in an XML schema language.
- The *platform-specific level* contains schemas which specify the semantics of each XML format in terms of the level above.
- The *platform-independent level* contains a conceptual schema which describes the information model of the system and covers the common semantics of all XML formats in the family in a uniform way.

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

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

In the rest of this section, we will describe particular horizontal levels in a more detail. We will also introduce a methodology which instructs users (XML schema designers) how to (1) build particular levels of the framework and (2) evolve the XML schemas using the framework. However, we do not delve into formal details. We refer to [61] for the formal background of the framework.

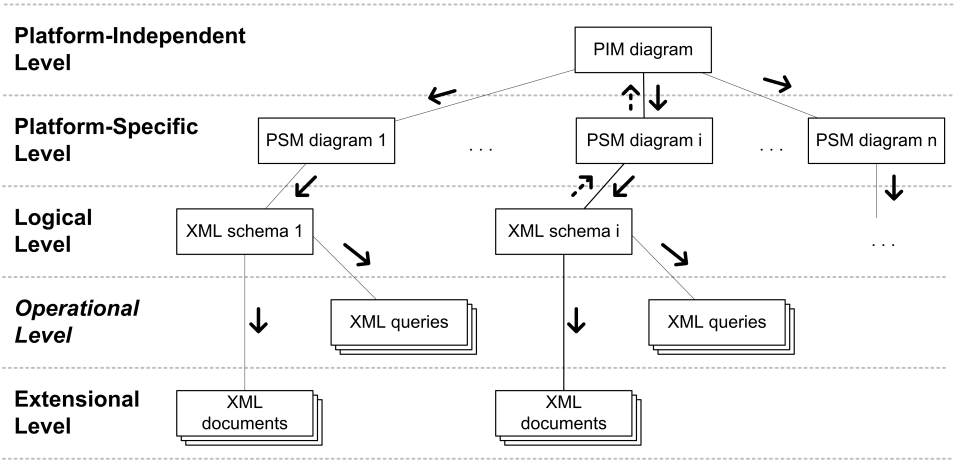


Figure 2.1: Five-level XML evolution framework

2.1 Framework Horizontal Levels

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

2.1.1 Logical, Operational and Extensional Level

The lowest level, called *extensional level*, represents particular XML schema instances that occur in the system. The instances are XML documents which are persistently stored in an XML database or exchanged between parts of the system or between the system and other systems as messages. The level one step higher, called *operational level*, represents operations over the instances. These might be, e.g., XML queries over the instances expressed in XQuery [6] or transformations of the instances expressed in XSLT [87]. The level above, called *logical level*, represents logical schemas that describe the structure of the instances. They are expressed in a selected XML schema language, e.g., XML Schema (XSD) [80], Relax NG [12], Schematron [29], etc. We demonstrate the three levels in Figure 2.2. It shows our two sample XML formats represented at the three levels.

There are two kinds of mappings between the three levels. There are mappings of instance XML documents to their XML schemas. The instances are XML documents *valid* against the XML schema. An instance XML element or attribute is mapped to its respective definition in the XML schema. The mapping is created automatically during XML document validation. For example, XML elements `cust` in the instances of XML format on the left of Figure 2.2 are mapped to the definition of the XML element `cust` in the XML schema. A valid instance is fully mapped to its XML schema.

The other kind are mappings of operations to XML schemas. Operations are based on the XPath language whose basic construct is a *path* comprising steps which select XML elements and attributes from the instance XML documents. The steps also specify required hierarchical relationships between the selected XML elements and attributes (e.g., parent/child or ancestor/descendant). A path is mapped to a respective chain of XML element or attribute definitions in the XML schema. For example, there are the following paths in the query for the XML format on the left of Figure 2.2: `//cust/hq` and `//cust/name`. They are mapped to the corresponding XML element definitions as depicted by the arrows.

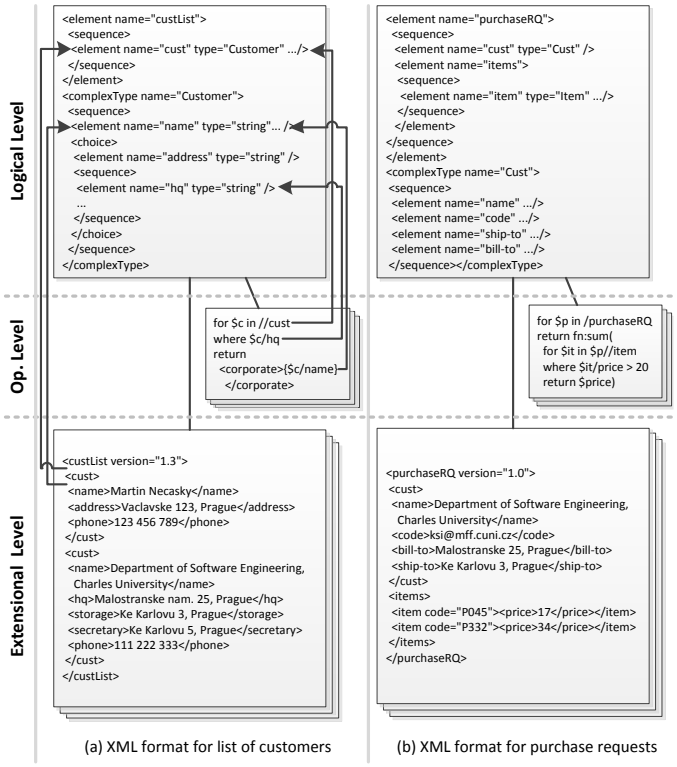


Figure 2.2: Two sample XML formats represented in the framework

Even these three levels indicate problems related to XML evolution. A change in one of them can trigger respective changes in the other two levels. Therefore, we need a mechanism which correctly propagates a change to the other levels. When the structure of an XML schema changes, its instances and related queries must be adapted accordingly so that their validity and correctness is preserved respectively. Some changes can be propagated automatically. However, there are also changes where automatic propagation is not always possible. For example, suppose that we want to split XML elements `name` in the left-hand side XML format from Figure 2.2 into two XML elements `firstname` and `lastname`. This change can be automatically propagated to the mentioned query (as a corresponding split of path `/cust/name` to two paths `/cust/firstname` and `/cust/lastname`). However, it cannot be automatically propagated to the instances until a designer specifies a function which splits a string with a full name to two substrings with first and last name. In general, such function does not exist. However, it may exist in some special cases, e.g., when full name strings have a strict structure consisting of first name followed by one white space followed by the last name. However, this depends on the designer whether (s)he is able to specify such function.

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

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

To overcome these problems, we introduce two additional levels to the framework briefly described at the beginning of this chapter. They represent two additional levels of abstraction of the XML schemas and are motivated by the MDA [52] principles. The topmost one is a *platform-independent level* which comprises a single conceptual schema in a *platform-independent model*. We will call it *PIM schema* and use the notation of UML class diagrams to express it. A sample PIM schema modeling the domain of customers and their purchases is depicted in Figure 2.3.

The level below is a *platform-specific level* which comprises an individual schema in a *platform-specific model* for each XML format. We will call it *PSM schema* and will also use the notation of UML class diagrams. However, we extended the notation in [61] so that it can be used for modeling XML formats. Two sample PSM schemas for our two XML formats are depicted in Figure 2.3.

A PSM schema models an XML format and can be viewed from two perspectives – *conceptual* and *grammatical*. The *conceptual perspective* models the semantics of the XML format in terms the PIM schema. The semantics is modeled as an unambiguous mapping of the components of the PSM schema to the components of the PIM schema. We demonstrate the mapping in Figure 2.3 on the right-hand side PSM schema. There is depicted the mapping of PSM class `PrivateCus` to PIM class `PrivateCus` and its PSM attributes `name`, `code`, `shipto` and `billto` to PIM attributes `name`, `code` and `address`, respectively. The PSM attributes `shipto` and `billto` are mapped to the same PIM attribute. Therefore, the semantics of the portion of the PSM schema is that `PrivateCus` class models a private customer with a name and code. Both shipping and billing address referred to in the purchase are the same address evidenced in the system for the customer. The PSM class `CorporeCus` is mapped similarly but its PSM attributes `shipto` and `billto` are mapped to the PSM attributes `storage` and `headquarters`, respectively. In other words, the semantics of `shipto` and `billto` attributes in the modeled XML formats is different for the private and corporate customers. Associations are mapped as well. For example, both PSM associations going to `PrivateCus` and `CorporateCus` are mapped to the PIM association connecting the PIM classes `Customer` and `Purchase`. There can also be components which are not mapped. They are displayed in the grey color, e.g., class `Items` or PSM associations `cust` and `items` in the PSM schema on the right. These components have no semantics.

2.1.2 Platform-Independent and Platform-Specific Levels

From the *grammatical perspective*, a PSM schema models a grammar of the respective XML format. In other words, it models the syntax of the XML format which is expressed at the logical level as an XML schema. The conversion of the PSM schema to a corresponding XML schema is automatic. Briefly, a class models a sequence of XML element and attribute declarations. An association

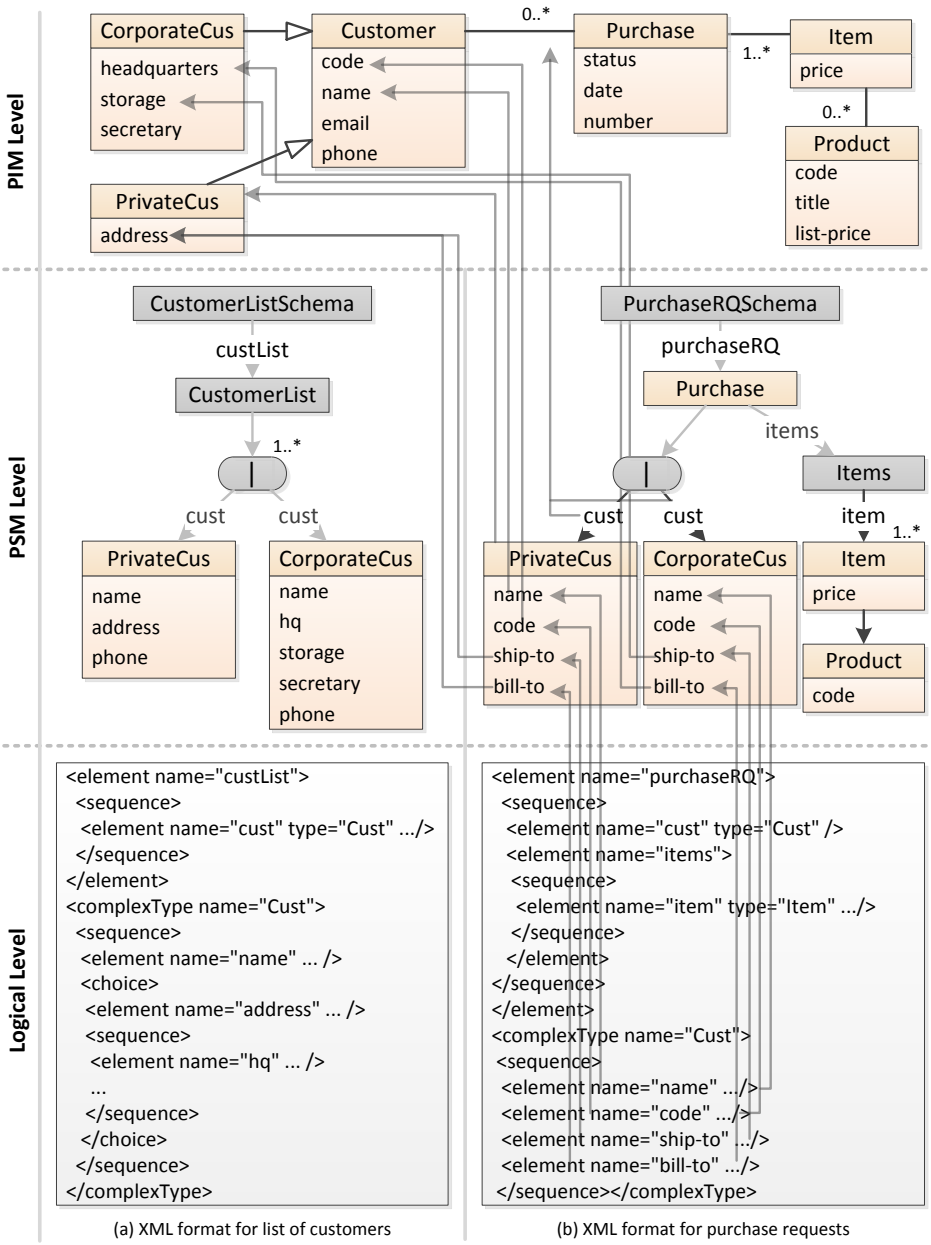


Figure 2.3: Two sample XML formats represented at logical, PSM and PIM levels

with a name models an XML element whose content is the sequence modeled by its child class.

Figure 2.3 also shows one of the extensions we introduced to the UML notation for the purposes of modeling XML schemas – choice content models. They are depicted as grey ovals with | symbol in the middle. A choice content model models a choice in the XML content. For example, the choice content model in the left-hand side schema in our example models a choice between two possible XML contents modelled by classes `PrivateCus` and `CorporateCus`. It is worth pointing out that the choice in the right-hand side schema is a choice between two different concepts (two different PSM classes mapped to two different PIM classes), but, from the grammatical perspective, there is no difference between the two (i.e., they both model exactly the same subtree). In fact, in the translation of the schema to an XSD, there is no `choice` used. Both variants are equivalent from

the grammatical perspective (but not from the conceptual, as we have shown, so we have to distinguish them in the PSM schema).

The complete framework forms a five-level hierarchy which interconnects all modeled XML formats in the family using the common PIM schema. The consistent evolution of the XML formats is realized using this common point. For instance, if a change occurs in a selected XML document, it is first propagated to the respective XML schema, PSM schema and, finally, to the PIM schema. We speak about an *upwards propagation*, in Figure 2.1 represented by the upwards arrows. It enables one to identify the part of the problem domain that was affected. Then, we can invoke the *downwards propagation*. It enables one to propagate the change of the problem domain to all the affected XML formats. In Figure 2.1 it is denoted by the downwards arrows.

Computation Independent Model MDA uses yet another level of abstraction above the PIM called the computation independent model (CIM). CIM is sometimes referred to as business model, because it uses the vocabulary of the particular problem domain and it should be understandable to subject matter experts of the problem domain. It is completely independent of the implementation.

In our framework, we do not support CIM explicitly and we will not refer to it in this thesis, however it can be used in combination with our framework.

2.2 Selected Part of the Problem

This thesis focuses on several parts (levels) of the five-level framework. We are adding integrity constraints into the framework. As we will show in the following chapters, integrity constraints can be defined both at the PIM and PSM levels and from the PSM level they can be automatically translated into expressions of the operational level.

We also target the problem of document adaptation, which in our five-level framework means propagating the changes from the PIM and PSM level into the extensional level (the documents in the system). Our adaptation algorithm uses schema comparison and for that we add versioning support into the model (effectively to the PIM and PSM levels).

3. Conceptual Model

In this chapter, we introduce the formal model of PIM and PSM schemas, as it was published previously [56]. In Section 3.3, we extend it with a formalism for the versioning of schemas. This formalism allows for modeling more versions of PIM and PSM schemas. We also show how concepts in different versions are connected and why we need such connections for document adaptation. The versioning extension was published in [46] and [45].

Both PIM and PSM schemas are UML class diagrams, but we introduce our own formal definitions of PIM and PSM schemas – this will allow us to focus on the core, which is required for XML modeling (classes, associations and attributes), without the need to concern with additional UML constructs, such as stereotypes, association classes, interfaces etc.

3.1 Formal Model of the PIM Level

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

- *a* Class $C \in \mathcal{S}_c$ *has a name assigned by function* name. *Inheritance of classes is modeled by partial function* isa, *which assigns a base class to a specific class. Cycles are forbidden in* isa *graph.*
- *an* Attribute $A \in \mathcal{S}_a$ *has a name, data type and cardinality assigned by functions* name, type, *and* card, *respectively. Moreover, A is associated with a class from* $\mathcal{S}_c$ *by function* class.
- *an* Association $R \in \mathcal{S}_r$ *is a set* $R = \{E_1, E_2\}$, *where* E_1 *and* E_2 *are members of the set of* association ends $\mathcal{S}_e$ *(*E_1 *and* E_2 *are called association ends of R). Both* E_1 *and* E_2 *have a cardinality assigned by function* card *and are associated with a class from* $\mathcal{S}_c$ *by function* participant. *We will call* $participant(E_1)$ *and* $participant(E_2)$ *participants of R. Association end has a name and a cardinality assigned by functions* name *and* card *respectively*[1].

For a class $C \in \mathcal{S}_c$, $attributes(C)$ *denotes the set of attributes of C and* $associations(C)$ *denotes the set of associations with C as a participant. A subset* $K \subseteq attributes(C)$ *can be declared a* key *for C.*

In the text, we will usually refer to the schema constructs using their name, i.e., when describing the schema in Fig. 2.3, when we write class `Purchase`, we speak about class $C \in \mathcal{S}_c$ s.t. $name(C) =$ *'Purchase'*. In the case of associations, we will refer to them by the name of the association end (if it is unambiguous) or by the names of the associated classes, e.g., `Purchase-Customer`. Note that an association is formally an unordered set of its two endpoints. In other words, associations are unordered in PIM schemas. Even though UML distinguishes directed associations we do not consider them in this thesis [2].

[1] When the name of an association end is the same as the name of the participant class (i.e., $name(E) = name(participant(E))$), we omit $name(E)$ in the figures. Similarly, we do not show cardinalities 1..1.

[2] Similarly we do not distinguish *composition* and *aggregation* in the framework.

As keys are concerned, the selection of attributes of each key is up to the modeller. (S)he should choose such a subset of class's attributes which uniquely identify an instance of the class. There can be more than one key for a class or none, if it is not important for the model.

In the example in Fig. 2.3, PIM schema is depicted at the top as a diagram. These are some of the constituents of the schema, according to Def. 1:

- $\mathcal{S}_c = \{\texttt{Customer}, \texttt{CorporateCus}, \texttt{Purchase}, \ldots\}$,
- $\mathcal{S}_a = \{\texttt{Customer.code}, \texttt{Item.price}, \ldots\}$,
- $\mathcal{S}_r = \{\texttt{Item-Product}, \texttt{Item-Purchase}, \ldots\}$.
- The schema also demonstrates inheritance:
  $\mathit{isa}(\texttt{CorporateCus}) = \mathit{isa}(\texttt{PrivateCus}) = \texttt{Customer}$.
- Types of attributes are not shown in the figure, but, e.g.,
  $\mathit{type}(\texttt{Purchase.date}) = \texttt{date}$, $\mathit{card}(\texttt{Purchase.date}) = 1$.
- For the association $\texttt{Customer-Purchase} = (E_{Customer}, E_{Purchase})$:
  $\mathit{participant}(E_{Customer}) = \texttt{Customer}$, $\mathit{participant}(E_{Purchase}) = \texttt{Purchase}$,
  $\mathit{name}(E_{Customer}) = \texttt{`Customer'}$, $\mathit{name}(E_{Purchase}) = \texttt{`Purchase'}$,
  $\mathit{card}(E_{Customer}) = 1$, $\mathit{card}(E_{Purchase}) = 0..*$.

3.2 Formal Model of the PSM Level

The purpose of the platform-specific model is to describe the system using constructs more tightly coupled to the selected platform and implementation technology, while reusing and referring to the general concepts defined at the PIM layer.

At the platform-specific level, we use slightly modified class diagrams. A PSM schema in our approach is a formal model which models the structure of XML documents of a given document type and also their semantics in terms of a PIM schema. A concrete PSM schema can be translated automatically to an XML schema written using one of the XML schema languages[3].

A distinctive feature of XML is its hierarchical structure – XML elements form a tree. This needs to be reflected in PSM. Thus, associations in our PSM schemas are all oriented and the schema has a distinctive skeleton – a forest.

Also, there are two ways how an atomic value can be stored in XML – as an element with a simple text content or an attribute. To distinguish these options, we introduce function *xform*, which specifies, whether an attribute of a class is mapped to an XML attribute or an XML element with a simple text content.

Finally, another feature of XML documents is certain variability – XML schema languages are usually based on regular tree grammars (RTG) [53] and thus provide means of choosing between several options in a certain location of the schema. For this purpose, we introduce the *content model*.

We define PSM schemas formally in Definition 2. In [61], we proved that PSM schemas have the expressive power of RTGs.

[3]We currently support export to XSD, Schematron and Relax NG.

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

- Class $C' \in \mathcal{S}'_c$ *has a name assigned by function name.* Content model $M' \in \mathcal{S}'_m$ *has a content model type assigned by function* cmtype. *cmtype*(M') $\in$ {`sequence`, `choice`, `set`}. *A class or content model can have a parent association assigned by partial function* parentAssociation *and a list of child associations assigned by function* childAssociations. *Classes/content models without parent association are called* root *classes/content models. Inheritance of classes is modeled by partial function* isa, *which assigns a base class to a specific class. Cycles are forbidden in* isa *graph.*
- Attribute $A' \in \mathcal{S}'_a$ *has a name, data type, cardinality and XML form assigned by functions* name, type, card *and* xform, *respectively.* *xform*(A') $\in \{e, a\}$. *Moreover, it is associated with a class from* $\mathcal{S}'_c$ *by function* class *and has a position assigned by function* position *within all the attributes associated with class*(A'). *The sequence of all the attributes of* C' *ordered by position will be denoted* attributes(C')
- Association $R' \in \mathcal{S}'_r$ *is a pair* $R' = (E'_1, E'_2)$, *where* E'_1, $E'_2 \in \mathcal{S}'_e$ (E'_1 *and* E'_2 *are called* association ends *of* R'). *Both* E'_1 *and* E'_2 *have a cardinality assigned by function* card *and each is associated with a class from* $\mathcal{S}'_c$ *or content model from* $\mathcal{S}'_m$ *assigned by function* participant. *We will use* *parent*(R') *to denote* *participant*(E'_1) *(called* parent *of* R'*) and* *child*(R') *to denote* *participant*(E'_2) *(called* child *of* R'*). Associations sharing a parent* C' *are ordered in the list* childAssociations(C'). *The index of the association in the list will be denoted* position$(R', \mathit{childAssociations}(C'))$. *Association ends have names, denoted* name(E'). *If* E' *is the parent end, its name is* `'parent'` *by default.*

We will use the notion PSM nodes $\mathcal{N}' = \mathcal{S}'_c \cup \mathcal{S}'_m$ *for the joined set of content models and classes. We also introduce the notion of* tree association *as the following subset of* $\mathcal{S}'_r$*:*
$\mathcal{S}'_r\bullet = \{R' = (E'_1, E'_2) \in \mathcal{S}'_r | R' = \mathit{parentAssociation}(\mathit{participant}(E'_2))\}$.
Classes, content models and tree associations form a forest. Formally, graph:
$\big(\mathcal{N}',\ \{(n_1, n_2) \in \mathcal{N}' x \mathcal{N}' \wedge (\exists (E'_1, E'_2) \in \mathcal{S}'_r\bullet)$
$(\mathit{participant}(E'_1) = n_1 \wedge \mathit{participant}(E'_2) = n_2)\}\big)$ *is a* forest *of trees rooted in* root *classes. Classes connected by an association to the schema class* $\mathcal{C}'_{\mathcal{S}'}$ *are called* top *classes.*

The main difference between the PIM schema and PSM schemas is that in PSM schemas, all associations are ordered and thus form an oriented graph with nodes from $\mathcal{N}'$ and $\mathcal{S}'_r$ as edges. Associations sharing a common parent are ordered (in other words, order is significant in the list of child associations of each node). If we remove the non-tree associations ($\mathcal{S}'_r \notin \mathcal{S}'_r\bullet$), we obtain a forest. Each class C' in this forest is either a *root* class or it has a particular association assigned – the *parentAssociation*.

In the figures, we usually do not show the names of the parent association ends (which by default is `'parent'`). The name shown for the association belongs to the child association end.

In the example in Fig. 2.3, PSM schemas are depicted in the middle as diagrams. These are some of the constituents of the first PSM schema, according to Def. 2:

- $\mathcal{C}'_{\mathcal{S}'} =$ `CustomerListSchema'`,
- $\mathcal{S}'_c = \{$`CustomerListSchema'`, `CustomerList'`, `PrivateCus'`, $\ldots\}$,
- $\mathcal{S}'_a = \{$`PrivateCus'.name'`, `CorporateCus'.phone'`, $\ldots\}$,
- $\mathcal{S}'_m = \{$`Choice'`$_1\}$
- $\mathcal{S}'_r = \{$`custList'`, `CustomerList'-Choice'`$_1$, $\ldots\}$.
- Types of attributes are not shown in the figure, but, e.g., $type($`PrivateCus'.name'`$) =$ `string`, $card($`PrivateCus'.name'`$) = 1$.
- For the association $R' =$ `CustomerList'-Choice'`$_1 = (E_{CustomerList'}, E_{Choice'_1})$:
  $participant(E_{CustomerList'}) =$ `CustomerList'`,
  $participant(E_{Choice'_1}) =$ `Choice'`$_1$,
  $name(E_{CustomerList'}) =$ `'parent'`, $name(E_{Choice'_1}) = \lambda$,
  $card(E_{CustomerList'}) = 1$, $card(E_{Choice'_1}) = 1..*$.
  $parent(R') = E_{CustomerList'}$, $child(R') = E_{Choice'_1}$
  $childAssociations($`Choice'`$_1) =$
  (`Choice'`$_1$`-PrivateCus'`, `Choice'`$_1$`-CorporateCus'`)
- top classes: {`CustomerList'`}

As we have already discussed in Section 2, we view a PSM schema from two perspectives – *conceptual* and *grammatical*. From the conceptual perspective, a PSM schema models the semantics of an XML format in terms of a PIM schema. This is formally expressed by mapping classes, attributes and associations in the PSM schema to their PIM equivalents. We call the mapping *interpretation of the PSM schema against the PIM schema*. The interpretation cannot be arbitrary. There are some restrictions which prevent from semantic inconsistencies. For example, a PSM attribute cannot be mapped to an arbitrary PIM attribute – the class of the PSM attribute must be mapped to the class of the PIM attribute. Without this condition, it would be possible to map, e.g., PSM attribute `code'` of PSM class `Customer'` in our sample PSM schema depicted in Figure 2.3 to PIM attribute `number` of PIM class `Purchase`. Intuitively, this does not make any sense, but our condition prevents from this mapping explicitly. It is forbidden because $class($`code'`$) =$ `Customer'` is mapped to `Customer` which is not the class $class($`number`$) =$ `Purchase`. In other words, the condition preserves semantic consistency between PSM and PIM attributes.

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

intuitively, does not make sense.

We will use $I(X') = X$ to denote that the interpretation of the PSM construct X' is X. E.g., $I($`PrivateCus'.ship-to'`$) =$ `PrivateCus.address`.

There can also be components which are not mapped to the PIM schema. These components are displayed in grey colour. We refer to [61] for detailed and formal description.

From the grammatical perspective, a PSM schema models the syntax of the XML format. In other words, it models the XML schema of the XML format. For example, the PSM schema depicted in Figure 2.3 models the syntax of the XML format whose instance is depicted in the same figure. The PSM schema does not depend on any particular XML schema language. It can be automatically translated to an arbitrary language. In [61] we showed how a PSM schema can be translated to a regular tree grammar [53] which can be expressed in XSD [55], Schematron [36] or RELAX NG.

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

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

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

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

Definition 2 introduces a PSM-specific construct called *content model*. By default, child associations of a class in a PSM schema model a sequentially ordered content. I.e., content modeled by child associations is propagated to the element

modeled by the class, with order preserved. However, XML schema languages are usually based on RTGs, which allow constructs for choosing from several variants of sub-content or declaring that the sub-contents may appear in any order. For these purposes, we use content models in PSM schemas – `choice` and `set`. `sequence` content model can be used to model a sequence of two choices etc.

In figures, the content models are displayed as small ovals with a specific symbol inside: "|" for choice, "{}" for set and "..." for sequence. Our sample PSM contains a choice content model. In this particular case it specifies that content of an XML element `cust` (modeled by the parent association of the choice content model) is one of the contents modeled by classes `PrivateCus'` and `CorporateCus'`.

Model Construct	**Modelled XML Construct**
$\mathcal{C}'_{\mathcal{S}'}$	Named child associations of the schema class $\mathcal{C}'_{\mathcal{S}'}$ model the allowed root XML elements. $\mathcal{C}'_{\mathcal{S}'}$ has no attributes by definition.
$C' \in \mathcal{S}'_c \setminus \mathcal{C}'_{\mathcal{S}'}$	A complex content which is a sequence of XML attributes and XML elements modelled by attributes in $attributes(C')$ followed by XML attributes and XML elements modelled by $childAssociations(C')$
$A' \in \mathcal{S}'_a$, s.t. $xform(A') = a$	An XML attribute with name $name(A')$, data type $type(A')$ and cardinality $card(A')$
$A' \in \mathcal{S}'_a$, s.t. $xform(A') = e$	An XML element with name $name(A')$, simple content with data type $type(A')$ and cardinality $card(A')$
$R' \in \mathcal{S}'_r$, s.t. $R' = (E'_1, E'_2)$, $name(E'_2) \neq \lambda$	An XML element with name $name(E'_2)$, complex content modelled by $child(R')$ and cardinality $card(E'_2)$. If $parent(R') = \mathcal{C}'_{\mathcal{S}'}$ then the XML element is the root XML element.
$R' \in \mathcal{S}'_r$, s.t. $R' = (E'_1, E'_2)$, $name(R') = \lambda$	Complex content modelled by $child(R')$

Table 3.1: XML modeled by PSM constructs

In [61] we have formally shown that the expressive power of our PSM schemas is the same as the expressive power of regular tree grammars (RTG) [53], i.e., each schema $\mathcal{S}'$ can be translated into a corresponding regular tree grammar $G_{\mathcal{S}'}$ and vice versa. This allows us to introduce the notion of validity of an XML document against a PSM schema.

Definition 3. *For a schema $\mathcal{S}'$, the set of conforming documents $\mathcal{T}(\mathcal{S}')$ equals to the language $\mathcal{L}(G_{\mathcal{S}'})$ generated by a grammar $G_{\mathcal{S}'}$. We will say that an XML document T is* valid *against $\mathcal{S}'$ if $T \in \mathcal{T}(\mathcal{S}') = \mathcal{L}(G_{\mathcal{S}'})$.*

PSM constructs – classes, content models, attributes and associations – from a PSM schema $\mathcal{S}'$ correspond to non-terminals $G_{\mathcal{S}'}$. A parser for grammar $G_{\mathcal{S}'}$ will take as an input a document T and decide whether $T \in \mathcal{L}(G_{\mathcal{S}'})$. If indeed $T \in \mathcal{L}(G_{\mathcal{S}'})$, during parsing, the parser will assign each element and attribute in T to some non-terminal corresponding to some PSM construct X'. We will call such elements/attributes *instances of X' in T*.

Normalization

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

Definition 4. *Let $\mathcal{S}'$ be a PSM schema. We call $\mathcal{S}'$* normalized PSM schema *when the following conditions are satisfied:*

$$(\forall R' \in childAssociations(C'_{\mathcal{S}'}))(name'(child(R')) \neq \lambda \wedge card'(child(R')) = 1..1) \quad (3.1)$$

$$(\forall R' \in \mathcal{S}'_r)(child'(R') \in \mathcal{S}'_m \rightarrow name'(child(R')) = \lambda) \quad (3.2)$$

$$(\forall M' \in \mathcal{S}'_m)(\exists R' \in \mathcal{S}'_r)(child'(R') = M') \quad (3.3)$$

If $\mathcal{S}'$ does not satisfy some of the conditions (3.1) – 3.1), it is called relaxed PSM schema.

The main purpose of PSM schema normalization is to remove redundancies and unreachable portions of the schema. A normalized schema is simpler than its relaxed equivalent. In [61], we presented, in the form of an algorithm, how every PSM schema $\mathcal{S}'$ can be normalized to schema $\overline{\mathcal{S}'}$ and proved formally that normalization does not reduce the modeled language (i.e., $\mathcal{L}(G_{\mathcal{S}'}) = \mathcal{L}(G_{\overline{\mathcal{S}'}})$). In this paper, we exploit these previous results and work only with normalized schemas. Therefore, the set of different types of possible edit operations we need to consider will be smaller (e.g., we do not have to consider an operation for moving a content model to a root) and, therefore, the introduced adaptation algorithms are less complex.

Condition (1) requires that an association with the parent being the schema class $C'_{\mathcal{S}'}$ has a named child end and that its cardinality is 1..1. This is natural because each such association models a root XML element. Therefore, its name needs to be specified and it has no sense to specify a cardinality different from 1..1. Condition (2) requires that an association end with a content model as a participant does not have a name. The names of child association ends specify element names. However, an association end with a content model as a participant does not model an element but only a part of the content of an element. Therefore, its name would not be used anyway. Condition (3) requires that each content model is a child of an association. A content model which is a root is unreachable in the schema and, therefore, redundant.

3.3 Versions of the Model

In this section, we add versioning features to the model, an extension we will use in Chapters 4 and 6.

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

properties). Then, the user can evolve $\tilde{v}$ to a desired state, but also go back to work with v or any other version existing in the system.

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

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

Definition 5. *Let $\mathcal{S}$ be a PIM schema and $\mathcal{S}'$ be a PSM schema. The set of all the components in $\mathcal{S}$ and $\mathcal{S}'$ will be denoted $\mathcal{S}_{all} = \mathcal{S}_c \cup \mathcal{S}_r \cup \mathcal{S}_a$ and $\mathcal{S}'_{all} = \mathcal{S}'_c \cup \mathcal{S}'_r \cup \mathcal{S}'_a$, respectively.*

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

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

Definition 6. *The function $ver : \mathcal{M}^* \to \mathcal{V}$ assigns a version to each PIM schema, PSM schema and to each of their components.*

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

Figure 3.1 shows two versions of a PIM schema and one PSM schema, XSDs for the schemas can be found in Appendix A.1 – A.2. The second version $\tilde{v}$ was created from the first version v using *branch* operation. Branch adds a new version to $\mathcal{V}$ (so that $\mathcal{V} = \{v, \tilde{v}\}$ and defines values of *ver* for the branched schemas and constructs. When system is branched, function *ver* would return v for the PIM and PSM schema on the left side and all their constructs. For the schemas on the right side and their constructs, it would return $\tilde{v}$. Each time a construct x is added to a schema S, $ver(x)$ is set to $ver(S)$. When a new PSM schema $\mathcal{S}'$ is mapped to a PIM schema $\mathcal{S}$, $ver(\mathcal{S}')$ is set to $ver(\mathcal{S})$. After being branched, the two versions can be edited separately. In the example, the user decided to reconnect PIM association `Address-Purchase` to `Address-Customer`, rename it from `delivered` to `has` and adapt the PSM schema accordingly.

Definition 7. *Let $\mathcal{S}^*$ be a set of PIM schemas and $\mathcal{S}^{*\prime}$ a set of PSM schemas. Let $\mathcal{K} \subseteq \mathcal{M}^*$ and $v \in \mathcal{V}$. We will use $\mathcal{K}[v] = \{C | ver(C) = v\}$ to denote* projection of K to version v *or simply* version projection*. In other words, version projection $\mathcal{K}[v]$ returns members of $\mathcal{K}$ that belong to version v.*

We require the following conditions to hold:

$$(\forall v \in \mathcal{V})(|\mathcal{S}^*[v]| = 1) \tag{3.4}$$

$$(\forall \mathcal{S} \in \mathcal{S}^*)(\mathcal{S}_{all} \subseteq \mathcal{M}^*[ver(\mathcal{S})]) \tag{3.5}$$

$$(\forall \mathcal{S}' \in \mathcal{S}^{*\prime})(\mathcal{S}'_{all} \subseteq \mathcal{M}^*[ver(\mathcal{S}')]) \tag{3.6}$$

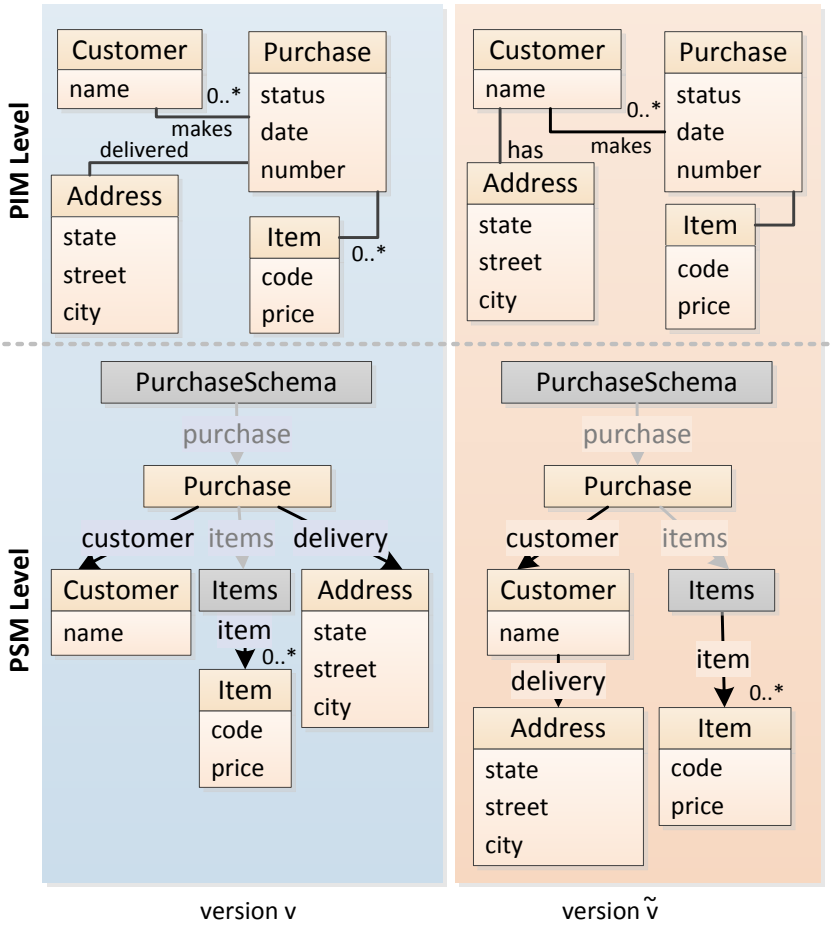


Figure 3.1: Two versions of a PIM and a PSM schema

The conditions in Definition 7 require that exactly one PIM schema exists for each version (3.4) and that all components of a given PIM or PSM schema belong to the same version (3.5, 3.6). We also require consistency in PSM – PIM mappings (*interpretations* of PSM constructs can be only the PIM constructs from the same version).

In our examples, we will usually show the system divided into version projections. Figure 3.1 shows the two version projections $\mathcal{M}^*[v]$ and $\mathcal{M}^*[\tilde{v}]$, $\mathcal{M}^*[v]$ on blue background, $\mathcal{M}^*[\tilde{v}]$ in orange background.

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

3.3.1 Document Adaptation in a Versioned System

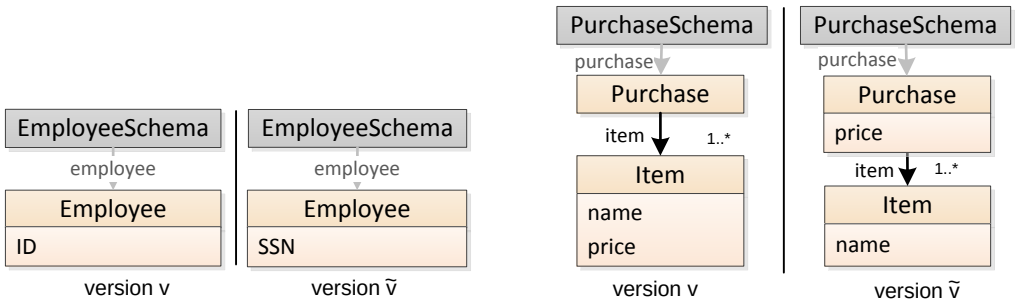
Document adaptation is a process triggered by schema evolution. By schema evolution we mean conducting certain operations upon the existing schema until reaching the desired final state – new version of the schema. Such a change can violate validity of instances of the schema, i.e. XML documents. The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

Definition 8. *We say that the set of conforming documents $\mathcal{T}(\mathcal{S}')$ of schema $\mathcal{S}'$ was* invalidated in the new version *(or just* invalidated*) if: $\exists T \in \mathcal{T}(\mathcal{S}') : T \notin$*

$\mathcal{T}(\widetilde{\mathcal{S}}')$. *If no such* T *exists, then* $\widetilde{\mathcal{S}}'$ *is called* backwards-compatible.

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

The document adaptation process starts with change detection – two schemas are compared and from the set of detected changes the system deduces steps required for successful document adaptation. Detecting changes in an XML schema or a model of an XML schema is not always straightforward; some differences between the old and new version can be interpreted in more than one way. For example, consider the PSM schemas in Figure 3.2.



(a) Employee schema (b) Purchase schema

Figure 3.2: Examples of schema evolution

There are two possible interpretations for evolution of schema in Figure 3.2a: either (1) attribute `ID` was removed and new attribute `SSN` was added or (2) attribute `ID` was renamed to `SSN`. When deciding which interpretation is the correct one, mapping to a PIM schema can be taken into account. E.g., if the attributes are mapped to PIM attributes p_1 and p_2, whereas p_2 is a new version of p_1, we can assume that the second interpretation is correct and the attribute was only renamed. But even this is still only a heuristic.

Likewise, there can be two interpretations of the change depicted in Figure 3.2b: (1) attribute `price` was moved from `Item` to `Purchase` or (2) the attribute was removed from `Item` and a new attribute was added to `Purchase`, having the same name coincidentally. The adaptation of the documents in this example is even more difficult to decide. In the second case, a correct value must be assigned to the new attribute. In the first case, the value of attribute `price` should be set to the sum of the values of `price` in all the items of the purchase.

These are examples of the problems that a non-trivial adaptation algorithm must solve. As we discuss in Chpt. 8, there are two fundamentally different classes of approaches to detecting changes between the two versions of the schema: either (1) recording the changes (as they are conducted during the schema evolution phase) or (2) by comparing the evolved schema to the original. A schema comparison approach must solve the problem of ambiguities of the sort as illustrated in the examples above.

If we do not want to settle for heuristics, the only correct solution is to find all possible interpretations and then let the user select the correct one. In our approach, we decided to solve this issue by adding another type of concepts into the model – *version links*. Version links connect constructs (i.e., classes, attributes etc.) that represent the same real-word concept in different versions of the model.

They work as a mapping between different versions of the schema and allow us to distinguish whether a given concept in the new version is a completely new concept or whether it is an update of an existing one. Therefore, it enables us to avoid heuristics.

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

Definition 9. *Let $\mathcal{S}^*$ be a set of PIM schemas, $\mathcal{S}^{*\prime}$ a set of PSM schemas.* A version links relation *is an equivalence relation $\mathcal{VL} \subset \mathcal{M}^* \times \mathcal{M}^*$ s.t. for any pair $(x, \tilde{x}) \in \mathcal{VL}$*

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

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

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

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

Definition 10. *Function $getInVer : (\mathcal{M}^* \times \mathcal{V}) \to \mathcal{M}^*$ is defined as follows:*

$$getInVer(x, v) = \tilde{x} \leftrightarrow (x, \tilde{x}) \in \mathcal{VL} \wedge ver(\tilde{x}) = v$$

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

Pairs for $\mathcal{VL}$ are also added during operation *branch* (a new version $\tilde{v}$ is created from version v), but can be changed by the user. Figure 3.3 shows the same schemas as Figure 3.1, this time with version links as they were created when the schemas were branched and the second schema was edited (links between attributes were omitted for clarity, also, the figure does not show two version links between the schemas themselves). All the version links in the figure were created and maintained by the system, without the need from the user to interfere.

When we go back to the pair of schemas depicted in Figure 3.2a, with version links, we can unambiguously decide, which interpretation is correct. The second

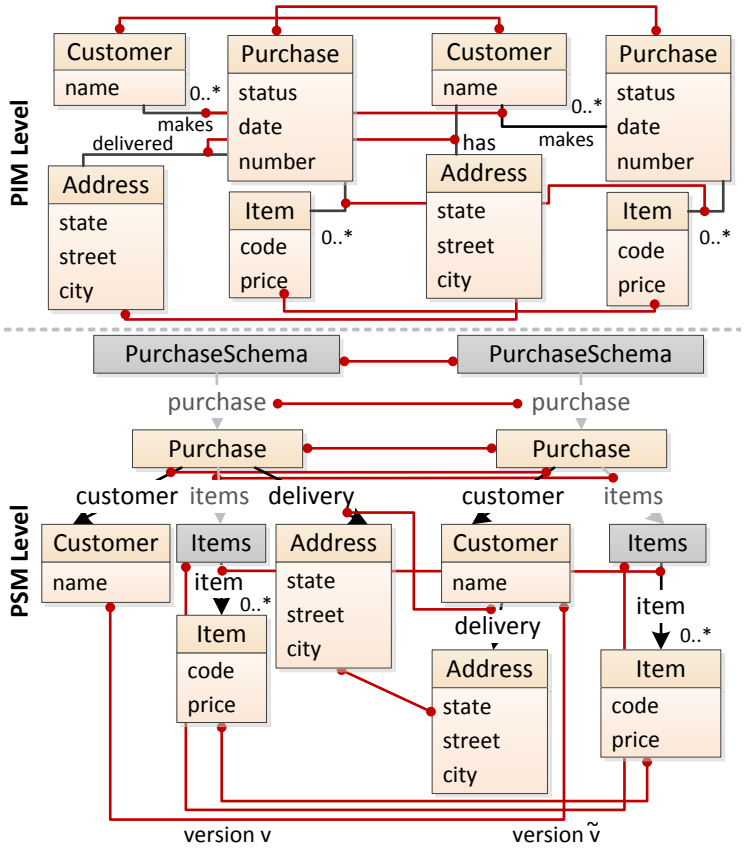


Figure 3.3: Two versions of a PIM and a PSM schema with version links visualized

interpretation (attribute `ID` was renamed to `SSN`) is correct when there exists a version link $(A', \widetilde{A'})$ (where A', $\widetilde{A'}$ are the attributes named `ID` and `SSN` respectively). The usual process leading to this situation would be that the user created new version of the schema via branch operation (which would create schema class $\widetilde{ES'}$ as a copy of schema class ES' named `EmployeeSchema`, class $\widetilde{E'}$ as a copy of class E' named `Employee`, association $\widetilde{AE'}$ as a copy of association AE' named `employee` and attribute $\widetilde{A'}$ as a copy of attribute A' named `ID`). Branch operation would create a copy of the schema and also version links $(ES', \widetilde{ES'})$, $(AE', \widetilde{AE'})$, $(E', \widetilde{E'})$, $(A', \widetilde{A'})$ between the copied concepts. Then the user would rename attribute $\widetilde{A'}$ from `ID` to `SSN` in the second version (but link $(A', \widetilde{A'})$ to the attribute in the first version would be preserved). For the second example in Figure 3.2b the situation is analogous and the interpretation would depend upon the existence of the version link between the attributes named `Item.price` and `Purchase.price` (moving an attribute also preserves the version link).

4. Document Adaptation

In this chapter, we specify the set of types of changes that our system can detect and we describe how changes are detected. For each type of change, we show how the change is adapted in the valid documents. Where several alternatives of adaptation exists for some change, we discuss the options. We also examine the changes from the perspective of backwards compatibility. Our document adaptation approach was published in [46] and [45]. Examples of issues we address can be found in Sec. 3.3.1.

4.1 Changes

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

As can be seen, we suppose that version links exist between two versions of a PSM schema. As we showed, this can be easily achieved when the user utilizes our framework which creates and maintains links as the user edits the versions. When a new version of the schema was not created using our framework (e.g., it was issued by a standardizing organization managing the specification which the modeled system adopted), the version links do not exist. To solve this problem, our framework supports reverse engineering and integration of schemas [34, 63]. It maps new PSM schemas or their new versions to the PIM schema. The version links can be then deduced by composing the interpretations of the versions of the PSM schema against the PIM schema. However, various heuristics together with broader user interaction is required to create relation $\mathcal{VL}$. The possibility to infer version links from heuristics is not studied in this thesis and is a part of our future work.

We can divide the set of types of changes which can occur between two versions of a PSM schema $\mathcal{S}'$ into four groups according to the character of a change (the classification is similar to [62]):

- addition – a new construct was added to $\mathcal{S}'$,
- removal – a construct was removed from $\mathcal{S}'$,
- migratory – a construct (and possibly its subtree in $\mathcal{S}'$) was moved to another part of $\mathcal{S}'$,
- sedentary – an existing construct in $\mathcal{S}'$ was adjusted in place, but not moved.

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

4.1.1 Change Predicates

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

Table 4.1 contains all the change predicates grouped by the scope with their respective categories and description. We suppose a PSM schema in two different versions $v, \tilde{v} \in \mathcal{V}$. We will use tilde to mark constructs that belong to $\mathcal{M}^*[\tilde{v}]$, constructs without the tilde mark belong to $\mathcal{M}^*[v]$. I.e., $\mathcal{S}' = (\mathcal{S}'_c, \mathcal{S}'_a, \mathcal{S}'_r, \mathcal{S}'_m, \mathcal{C}'_{\mathcal{S}'}) \in \mathcal{M}^*[v]$ denotes the PSM schema in version v and $\widetilde{\mathcal{S}'} = (\widetilde{\mathcal{S}'_c}, \widetilde{\mathcal{S}'_a}, \widetilde{\mathcal{S}'_r}, \widetilde{\mathcal{S}'_m}, \widetilde{\mathcal{C}'_{\mathcal{S}'}}) \in \mathcal{M}^*[\tilde{v}]$ denotes the PSM schema in version $\tilde{v}$.

For an example of change predicates, let us go back to Figure 3.2. Let us assume a version link between attributes A' named `ID` and $\widetilde{A'}$ named `SSN` in Figure 3.2a. Table 4.1 contains a change predicate *attributeRenamed* with parameters $\widetilde{A'} \in \widetilde{\mathcal{S}'_a}$ and $\tilde{n'}$ (the new name). Statement $(\widetilde{A'}, \text{"}SSN\text{"}) \in attributeRenamed_{ins}$ is a formal expression of the fact that the name of the attribute was changed from `ID` to `SSN`.

Similarly, let us assume a version link between A'_p and $\widetilde{A'_p}$ (i.e., attributes `Item.price` and `Purchase.price` in Figure 3.2b, where $ver(A'_p) = v$ and $ver(\widetilde{A'_p}) = \tilde{v}$). Let $\widetilde{C'_p}$ be the class named `Purchase` in version $\tilde{v}$. Table 4.1 contains a change predicate *attributeMoved* with parameters $\widetilde{A'} \in \widetilde{\mathcal{S}'_a}$, $\widetilde{C'_n} \in \widetilde{\mathcal{S}'_c}$ and $\tilde{i'} \in \mathbb{N}_0$. Statement $(\widetilde{A'_p}, \widetilde{C'_p}, 0) \in attributeMoved_{ins}$ is a formal expression of the fact that attribute `price` was moved to class `Purchase` to the position 0.

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

$$\begin{aligned}&\widetilde{A'} \in \widetilde{\mathcal{S}'_a} \wedge \widetilde{C'} \in \widetilde{\mathcal{S}'_c} \setminus \{\widetilde{\mathcal{C}'_{\mathcal{S}'}}\} \wedge \tilde{i'} \in \mathbb{N}_0 \wedge getInVer(\widetilde{A'}, v) = \perp \\ &\quad \wedge position(\widetilde{A'}, attributes(\widetilde{C'})) = \tilde{i'} \leftrightarrow attributeAdded(\widetilde{A'}, \widetilde{C'}, \tilde{i'})\end{aligned} \tag{4.1}$$

$$\begin{aligned}&\big(\widetilde{R'} \in \widetilde{\mathcal{S}'_r} \wedge \tilde{i'} \in \mathbb{N}_0 \wedge C'_1 \in \mathcal{N}' \wedge \widetilde{C'_1} \in \widetilde{\mathcal{N}'} \wedge \\ &\quad getInVer(\widetilde{R'}, v) \neq \perp \wedge C'_1 = getInVer(\widetilde{C'_1}, v) \wedge \\ &\quad parent(getInVer(\widetilde{R'}, v)) = C'_1 = getInVer(parent(\widetilde{R'}), v) \wedge \\ &\quad \tilde{i'} \neq position(R', childAssociations(C'_1)) \wedge \\ &\quad \tilde{i'} = position(\widetilde{R'}, childAssociations(\widetilde{C'_1}))\big) \\ &\quad \leftrightarrow associationPositionChanged(\widetilde{R'}, \tilde{i'})\end{aligned} \tag{4.2}$$

Change predicate	Category	Description
$classAdded(\widetilde{C'}, \widetilde{R'})$	Addition	A new class $\widetilde{C'}$ is added as a child of association $\widetilde{R'}$ (if $\widetilde{R'} = \bot$, $\widetilde{C'}$ is added as a new root class).
$classRemoved(C')$	Removal	Class C' is removed.
$classRenamed(\widetilde{C'}, \tilde{n'})$	Sedentary	The name of class $\widetilde{C'}$ is changed to $\tilde{n'}$. The name is mandatory for PSM classes, but can be changed.
$classMoved(\widetilde{C'}, \widetilde{R'_n})$	Migratory	Class $\widetilde{C'}$ is moved and becomes a child of association $\widetilde{R'_n}$ in version $\tilde{v}$ (or becomes a new root class, in that case $\widetilde{R'_n} = \bot$). This change encompasses changes of the *child* participant of associations (in contrast to *associationMoved* – see below).
$attributeAdded(\widetilde{A'}, \widetilde{C'}, \tilde{i'})$	Addition	A new attribute $\widetilde{A'}$ is added to class $\widetilde{C'}$ at position $\tilde{i'} \in \mathbb{N}_0$.
$attributeRemoved(A')$	Removal	Attribute A' is removed.
$attributeRenamed(\widetilde{A'}, \tilde{n'})$	Sedentary	The name of attribute $\widetilde{A'}$ is changed to $\tilde{n'}$.
$attributeMoved(\widetilde{A'}, \widetilde{C'_n}, \tilde{i'})$	Migratory	The value of $class(\widetilde{A'})$ is changed, i.e., attribute $\widetilde{A'}$ is moved from class C'_o to class $\widetilde{C'_n}$ at position $\tilde{i'} \in \mathbb{N}_0$. Moves within the same class are detected by *attributeIndexChanged*.
$attributeXFormChanged(\widetilde{A'}, \tilde{f'})$	Sedentary	The value of *xform* is changed from a to e or vice versa for attribute $\widetilde{A'}(\tilde{f'} \in \{a, e\})$.
$attributeTypeChanged(\widetilde{A'}, \widetilde{D'})$	Sedentary	The type of attribute $\widetilde{A'}$ is changed to $\widetilde{D'} \in \mathcal{D}$.
$attributeIndexChanged(\widetilde{A'}, \tilde{i'})$	Migratory	Attribute $\widetilde{A'}$ is moved to position $\tilde{i'} \in \mathbb{N}_0$ within the same class as in version v. Moves between classes are detected by *attributeMoved*.
$attributeCardinalityChanged(\widetilde{A'}, \tilde{c'})$	Sedentary	The cardinality of attribute $\widetilde{A'}$ is changed to $\tilde{c'}$.
$associationAdded(\widetilde{R'}, \widetilde{C'}, \tilde{i'})$	Addition	A new association $\widetilde{R'}$ is added to the content of class $\widetilde{C'}$ at position $\tilde{i'} \in \mathbb{N}_0$.
$associationRemoved(R')$	Removal	Association R' is removed.
$associationEndRenamed(\widetilde{E'}, \tilde{n'})$	Sedentary	The name of the association end $\widetilde{E'}$ is changed to $\tilde{n'}$.
$associationMoved(\widetilde{R'}, \widetilde{P'_n}, \tilde{i'})$	Migratory	Association $\widetilde{R'}$ is moved from the content of node P'_o to the content of node $\widetilde{P'_n}$ at position $\tilde{i'} \in \mathbb{N}_0$. This change encompasses changes of the *parent* participant of associations (in contrast to *classMoved* and *contentModelMoved* – see below).
$associationEndCardinalityChanged(\widetilde{E'}, \tilde{c'})$	Sedentary	The cardinality of association end $\widetilde{E'}$ is changed to $\tilde{c'}$.
$associationPositionChanged(\widetilde{R'}, \tilde{i'})$	Migratory	Association $\widetilde{R'}$ is moved to position $\tilde{i'} \in \mathbb{N}_0$ (within *childAssociations* of the same class as in version v).

Table 4.1: Classification of changes 1/2

Change predicate	Category	Description
$contentModelAdded(\widetilde{M'}, \widetilde{R'})$	Addition	A new content model $\widetilde{M'}$ is added as a child of association $\widetilde{R'}$.
$contentModelRemoved(M')$	Removal	Content model M' is removed.
$contentModelMoved(\widetilde{M'}, \widetilde{R'_n})$	Migratory	Content model $\widetilde{M'}$ is moved and becomes a child of association $\widetilde{R'_n}$ in version $\tilde{v}$. Content models cannot be roots in a normalized PSM schema (see Definition 4). Thus, unlike *classMoved*, $\widetilde{R'_n}$ is never null for *contentModelMoved*.
$contentModelTypeChanged(\widetilde{M'}, \tilde{t'})$	Sedentary	The type of content model (`sequence`, `set`, `choice`) $\widetilde{M'}$ is changed to $\tilde{t'} \in \{\texttt{sequence}, \texttt{set}, \texttt{choice}\}$.

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

Table 4.1: Classification of changes 2/2

$$\begin{aligned}\widetilde{C'} \in \widetilde{\mathcal{S}}'_c \setminus \{\widetilde{\mathcal{C}'_{\mathcal{S}'}}\} \wedge \widetilde{R'_n} \in \widetilde{\mathcal{S}}'_r \wedge getInVer(\widetilde{C'}, v) = C' \neq \perp \wedge child(\widetilde{R'_n}) = \widetilde{C'} \wedge \\ \big[(\exists R'_o \in \mathcal{S}'_r)(child(R'_o) = C' \wedge \\ R'_o \neq getInVer(\widetilde{R'_n}, v)) \vee (\forall R'_o \in \mathcal{S}'_r)(child(R'_o) \neq C')\big] \\ \leftrightarrow classMoved(\widetilde{C'}, \widetilde{R'_n}))\end{aligned} \tag{4.3}$$

$$\begin{aligned}\widetilde{C'} \in \widetilde{\mathcal{S}}'_c \setminus \{\widetilde{\mathcal{C}'_{\mathcal{S}'}}\} \wedge getInVer(\widetilde{C'}, v) = C' \neq \perp \\ \wedge (\exists R'_o \in \mathcal{S}'_r)(child(R'_o) = C') \wedge (\forall \widetilde{R'_n} \in \widetilde{\mathcal{S}}'_r)(child(\widetilde{R'_n}) \neq \widetilde{C'}) \\ \leftrightarrow classMoved(\widetilde{C'}, \perp))\end{aligned} \tag{4.4}$$

Predicate (4.1) says that the examined attribute $\widetilde{A'}$ has no counterpart A' present in version v and the position of $\widetilde{A'}$ among the attributes of class $\widetilde{C'}$ equals to $\tilde{i'}$. Predicate (4.2) says that the parent of the examined association $\widetilde{R'}$ has not changed between versions v and $\tilde{v}$, but the position of $\widetilde{R'}$ in the content of its parent has changed to $\tilde{i'}$. Predicate (4.3) says that the examined class $\widetilde{C'}$ was moved under association $\widetilde{R'_n}$ either from the root or from another association, whereas Predicate (4.4) says that $\widetilde{C'}$ was moved from an association R'_o to a root.

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

subroutine for each change predicate. For instance, for predicate *classMoved*, it is not necessary to test all associations for parameter $\widetilde{R'_n}$, but only the actual parent of class $\widetilde{C'}$. That is how the actual implementation works.

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

Algorithm 1 DetectChanges

Input: old and new version $v, \tilde{v} \in \mathcal{V}$, PSM schemas $\mathcal{S}'$, $\widetilde{\mathcal{S}}'$
Output: $C_{\mathcal{S}',\widetilde{\mathcal{S}}',v,\tilde{v}}$ – set of changes between $\mathcal{S}'$ and $\widetilde{\mathcal{S}}'$

```
1: C_{S',S̃',v,ṽ} ← ∅
2: for all change predicate c do
3:    c_ins ← ∅
4: end for
5: for all change predicate c of arity k do
6:    for all tuple t ∈ (S'_all × S̃'_all)^k do
7:       if c(t) then {tuple t satisfies c}
8:          c_ins ← c_ins ∪ t
9:       end if
10:   end for
11:   C_{S',S̃',v,ṽ} ← C_{S',S̃',v,ṽ} ∪ {(c, c_ins)}
12: end for
```

Inheritance Algorithm 1 does not detect changes in PSM inheritance, but it could be readily extended. From the document adaptation point of view, inheritance means:

1. allowing content from the base class in the instance of the specific class
2. allowing instances of inherited classes in the place where an instance of the base class is allowed

The first property can be achieved by treating each specific class $\widetilde{C'_s}$ as if it had another unnamed association preceding the rest of the associations in $childAssociations(\widetilde{C'_s})$ and putting the inherited attributes from the base class $\widetilde{C'_g}$ before the attributes of $\widetilde{C'_s}$ itself. The second property can be achieved by treating each association $\widetilde{R'}$ s.t. $child(\widetilde{R'}) = \widetilde{C'_g}$ as if there was leading to a `choice` content model $\widetilde{N'}$ and $\widetilde{N'}$ had unnamed associations leading to all the classes $\widetilde{C'_{s_i}}$, which inherit from $\widetilde{C'_g}$.

4.1.2 Impact on Validity

The output set $C_{\mathcal{S}',\widetilde{\mathcal{S}}',v,\tilde{v}}$ of algorithm *DetectChanges* can be further analyzed. Having detected the set of change instances $C_{\mathcal{S}',\widetilde{\mathcal{S}}',v,\tilde{v}}$, we can determine the impact of evolution on validity. Some change instances do not affect validity of the documents in $\mathcal{T}(\mathcal{S}')$. However, they may affect other parts of our five-level framework. For example, during the translation of the PSM schema into XSD, class

names are used for naming the generated complex types. Renaming a class does not invalidate $\mathcal{T}(\mathcal{S}')$ (because class names do not correspond to content of the documents), but the XSD generated from $\mathcal{S}'$ will differ from the XSD translated from $\widetilde{\mathcal{S}}'$.

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

Definition 11. *Let c be a change predicate (as listed in Table 4.1) and $i_c \in c_{ins}$ its instance. We define predicate c^{NI}, called* NI-predicate for *c, with the same parameters as change predicate c. When c^{NI} is satisfied for an instance i_c, this instance does not affect validity of documents in $\mathcal{T}(\mathcal{S}')$.*

Example 4.1.2 shows several NI-predicates and Lemma 1 joins NI-predicates with the notion of backwards compatibility from Definition 8. Its proof is a direct application of the definitions.

As an example consider the following NI-predicates:

$$classRenamed^{NI}(\widetilde{C'}, \tilde{n'}) \leftrightarrow true \tag{4.5}$$

$$attributeCardinalityChanged^{NI}(\widetilde{A'}, \tilde{m'}..\tilde{n'}) \leftrightarrow$$
$$getInVer(\widetilde{A'}, v) = A' \ \wedge card(A') = m'..n' \ \wedge m' \geq \tilde{m'} \ \wedge n' \leq \tilde{n'} \tag{4.6}$$

$$associationPositionChanged(\widetilde{R'}, \tilde{i'}) \leftrightarrow$$
$$getInVer(\widetilde{R'}, v) = R' \ \wedge parent(\widetilde{R'}) = \widetilde{M'} \in \widetilde{\mathcal{S}'_m} \ \wedge parent(R') = M' \in \mathcal{S}'_m \ \wedge$$
$$\big((cmtype(M') \ \in \{\texttt{sequence}, \texttt{set}\} \ \wedge cmtype(\widetilde{M'}) = \texttt{set}) \ \vee$$
$$(cmtype(M') = \texttt{choice} \ \wedge cmtype(\widetilde{M'}) = \texttt{choice})\big) \tag{4.7}$$

Predicate (4.5) is satisfied for all instances of *classRenamed*, because the name of a class does not correspond to any part of the modeled XML document. All instances of *classRenamed* thus do not violate validity. Predicate (4.6) is satisfied for those instances of *attributeCardinalityChanged* which broaden the cardinality interval. Predicate (4.7) is satisfied for those instances of *associationPositionChanged* which reorder content of content models of type `set` and `choice`. For these, the ordering of content is not significant.

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

Lemma 1. *Let $\mathcal{S}'$ and $\widetilde{\mathcal{S}}'$ be two versions of a PSM schema ($ver(\mathcal{S}') = v$, $ver(\widetilde{\mathcal{S}}') = \tilde{v}$). Let $C_{\mathcal{S}',\widetilde{\mathcal{S}}',v,\tilde{v}}$ be the output set of algorithm* DetectChanges *for these schemas. Then:*

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

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

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

It must be emphasized that NI-predicates can not decide backwards compatibility by themselves. It is possible that the schema is backwards compatible, even thought some change predicates are violated (i.e., the implication in Lemma 1 can not be turned into an equivalence). Change predicates are always 'local' and they do not consider other changes in the schema (i.e., change predicates for two change instances may be violated, but the two changes combined are backwards compatible). Testing backwards compatibility exhaustively would mean to decide whether $\mathcal{L}(G_{\mathcal{S}'}) \subseteq \mathcal{L}(G_{\widetilde{\mathcal{S}'}})$ which is generally undecidable.

4.2 Adaptation

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

$$\forall T \in \mathcal{T}(\mathcal{S}') \setminus \mathcal{T}(\widetilde{\mathcal{S}'}) : adapt(T) \in \mathcal{T}(\widetilde{\mathcal{S}'}) \tag{4.8}$$

$$\forall c, \forall i, i \in c_{ins}, (c, c_{ins}) \in C_{\mathcal{S}', \widetilde{\mathcal{S}'}, v, \widetilde{v}} : \text{instance } i \text{ is adapted correctly in } adapt(T) \tag{4.9}$$

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

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

We do not expect any specific implementation language [87, 91, 81, 28] in the rest of this section, however, our implementation uses XSLT. We will describe how our framework generates the adaptation script in XSLT in Sec. 4.3.

4.2.1 Class Changes

classAdded $(\widetilde{C'}, \widetilde{R'})$ If the added class $\widetilde{C'}$ is a *top* class (i.e., child of schema class $\mathcal{C}'_{\mathcal{S}'}$), Definition 4 requires the association between $\mathcal{C}'_{\mathcal{S}'}$ and $\widetilde{C'}$ to have a name. New named top class models a candidate for a root node of the document. Adding

a new such candidate does not require adaptation of the existing documents. Similarly, when $\widetilde{C'}$ is not a top class, but $card(\widetilde{R'}) = 0..n$ (the XML content modelled by the association and the class is optional) or $parent(R') = X' \in \mathcal{S}'_m$ (class is added into a content model) and the content model is a choice or set, the change does not require adaptation (the added class defines a new *optional* part of a document).

In other cases, an instance of class $\widetilde{C'}$ must be created during adaptation. Creating new content is a problem of its own and we will analyse it later in this chapter. If the $card(\widetilde{R'}) = 1..1$ and $name(\widetilde{C'}) \neq \lambda$, the adaptation algorithm can create a new XML element. However, when $card(\widetilde{R'}) = m..n$ where $m < n$, the algorithm must decide (or ask the user) how many instances it should create. If it creates more then one instance and some constructs were moved under $\widetilde{C'}$, again, the algorithm must decide how to distribute the moved content into the created instances of $\widetilde{C'}$.

classRemoved (C') Removal of a construct from the model must be always solved by removal of the content modeled by the removed construct from the documents in $\mathcal{T}(\mathcal{S}')$ (removing the instances). However, the content modeled by the whole subtree each instance cannot be instantly removed from the document, because some other changes may move parts of this content to other parts of the document.

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

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

The remaining *classMoved* change will be discussed in Sec. 4.2.5.

4.2.2 Attribute Changes

attributeAdded $(\widetilde{A'}, \widetilde{C'}, \tilde{i'})$ If attribute $\widetilde{A'}$ is added as mandatory, a new content must be added into the document – either an XML element with a simple content or an XML attribute (if $xform(\widetilde{A'}) = e$ or a respectively). The complications are similar as in the case of *classAdded* change.

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

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

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

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

attributeTypeChanged $(\widetilde{A'}, \widetilde{D'})$ Let D' be the type in version v, i.e., $D' = type(getInVer(\widetilde{A'}, v))$ and $dom(D')$ its domain. Adaptation of documents may be skipped in case when $dom(D') \subseteq dom(\widetilde{D'})$. This condition is guaranteed if D' is a type derived from $\widetilde{D'}$ using restriction in the XSD type system (see [5]). The condition means that the requirements for the documents were relaxed and a more general set of values is allowed. In the opposite situation, instead of a general set of values, the 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 new version. In such case $D' \supseteq \widetilde{D'}$. In the general case the two sets are incomparable.

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

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

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

Alternatively, the function can be defined separately attribute $\widetilde{A'}$ with changed type.

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

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

attributes ($\widetilde{C'}$).

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

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

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

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

When adding nodes, the values for these nodes must be assigned. Raising the lower cardinality from $m' \geq 1$ to $\tilde{m'} > m'$ raises the minimum allowed occurrences, (the special case $m' = 0$ and $\tilde{m'} \geq 1$ makes an optional subelement/attribute mandatory). That is why approaches to generate values of attributes need to be discussed. In this particular case, a simple solution would be using a default value of the attribute – see Section 4.2.6 for more details.

attributeMoved **($\widetilde{A'}$, $\widetilde{C'_n}$, $\tilde{i'}$)** Moving an attribute is an evolution operation that requires more in-depth enquiry. The aim of our approach is to keep the semantics of the adapted document and not to loose the existing data during adaptation. The trivial propagation solution – deleting the attribute from its former location in the document and creating a new attribute in the new location (as used in [77] and [24]) – is not suitable, because the value of the attribute is lost.

The straightforward solution is to create the instance at the new location, copy the value from the old location and remove the old instance. This is suitable in many cases, but when there are cardinalities involved, the situation gets complicated. The problem is, that there may be several instances of the A' in the old version and the number of required instances of $\widetilde{A'}$ in the new version is different.

The most general approach is to couple each instance $(\widetilde{A'}, \widetilde{C'_n}, \tilde{i'})$ of *attributeMoved* change with an adaptation function $attMove_{\widetilde{A'}}(oldInstances, newInstance)$ where *oldInstances* selects all existing instances of A' and *newInstance* contains the new location of the instance in the adapted document. The result of the function is the new value for the instance of $\widetilde{A'}$ in the new document. In general, the function $attMove_{\widetilde{A'}}$ is defined by the user, but the system can provide the user with a suggestion in certain cases – several types of the most common scenarios can be distinguished.

In the following text we expect that the attribute was moved between classes C'_o and $\widetilde{C'_n}$, i.e., attribute $A' \in attributes(C'_o)$, $\widetilde{A'} \in attributes(\widetilde{C'_n})$. Let $\widetilde{C'_o} = getInVer(C'_o, \tilde{v})$ and $C'_n = getInVer(\widetilde{C'_n}, v)$ be the new version of class C'_o and the

old version of class $\widetilde{C'_n}$ respectively, both can be $\perp$. In the situation depicted in Figure 3.2b $A' = \texttt{price}_1, \widetilde{A'} = \texttt{price}_2, C'_o = \texttt{Item}_1, C'_n = \texttt{Purchase}_1, \widetilde{C'_o} = \texttt{Item}_2, \widetilde{C'_n} = \texttt{Purchase}_2$ (we use subscripts to distinguish constructs in version v and $\tilde{v}$). We will use an auxiliary function *tree*, that returns the smallest tree that contains a set of nodes. Formally:

Definition 12. *For a rooted tree (V, E), $tree(X)$, $tree : 2^V \rightarrow 2^{(V \cup E)}$ returns the nodes and edges of the subgraph of the smallest common subtree for a set $X \subseteq V$, containing root b of the common subtree, members of X and for each $n \in X$ path between n and the root b.*

An example of the result of function *tree* is depicted in Figure 4.1. The result of $tree(\{f, g, i\})$ is the set $\{a, b, d, f, g, i, a-b, b-f, f-i, a-d, d-g\}$, where a is the root of the common subtree and $a-b$ the edge from a to b, etc.

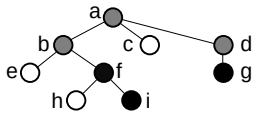


Figure 4.1: Example of *tree* function

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

The intuitive meaning of predicate $stable(\mathcal{X}')$ is that there were no radical changes in the structure of the schema concerning the members of $\mathcal{X}'$.

In the case of $C'_n \neq \perp, T' = tree(\{C'_o, C'_n\})$ and $stable(T')$ holds, we can suggest the following adaptations, which cover the usual scenarios:

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

– If C'_o is an ancestor of C'_n in the PSM tree, $card(A') = m'..*$ and $card(\widetilde{A'}) = \widetilde{m'}..1$, then this is an inverse case to the one above. The respective values of A' can be distributed to the new locations. Nonetheless, a user may have to specify the distribution precisely.

When none of the conditions above is satisfied, a possible general approach is to use the function $attMove_{\widetilde{A'}} = identityN$ which returns the value of the n-th instance of A' when required at the n-th location in the adapted document. Possible other generic $attMove_{\widetilde{A'}}$ functions may be provided by the framework and the framework may also allow the user to create new ones. New functions could be defined either in an implementation language (such as XSLT), but this solution is not in concord with the objective to abstract the user from working with the implementation language directly. In Chpt. 6, we will show a different solution which uses expression languages.

4.2.3 Association Changes

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

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

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

associationEndCardinalityChanged **($\widetilde{E'}$, $\tilde{c'}$)** Changing the cardinality of the parent node does not require any revalidation. So, let us assume that $\widetilde{E'}$ is the child node of association $\widetilde{R'}$. Similarly as with *attributeCardinalityChanged*, there exist two adaptation actions, from which none, one or both must be undertaken to adapt a document (varying from document to document).

Let $card(E') = m'..n'$ and $card(\widetilde{E'}) = (\tilde{m'}, \tilde{n'})$.

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

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

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

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

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

When adding, the content for the new instances must be generated (this involves generating a whole XML subtree).

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

In the case numbered (2), no adaptation is necessary, because the ordering of associations in choices/sets has no effect on validity of documents.

associationEndRenamed $(\widetilde{E'}, \tilde{n'})$ If $\widetilde{E'}$ is the parent association end, no adaptation action is necessary, because the name of the parent association end is not reflected anywhere in the XML documents.

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

- If $n' = \lambda \wedge \tilde{n'} \neq \lambda$, the association was given a name, which means the each instance (since E' can have cardinality $\neq 1..1$) will be wrapped in a new XML element with name $\tilde{n'}$. If the subtree of $participant(E')$ models some attributes with $xform = a$, which are not in a subtree of another class with non-empty name, the instances of these attributes will now be moved to newly created the wrapping XML element.
- If $n' \neq \lambda \wedge \tilde{n'} \neq \lambda$, the association is renamed, which means each wrapping XML element modelled by R'will be renamed to $\tilde{n'}$.
- If $n' \neq \lambda \wedge \tilde{n'} = \lambda$, the name is removed from an association, the needed adaptation is an exact opposite of the first case, which means that the wrapping XML element is removed (and when it contains some XML attributes they are moved upwards).

The remaining *associationMoved* change will be discussed in the following Sec. 4.2.5.

4.2.4 Content Model Changes

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

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

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

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

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

The remaining *contentModelMoved* change will be discussed in the following Sec. 4.2.5.

4.2.5 Changes Moving Classes, Content Models and Associations

associationMoved $(\widetilde{R'}, \widetilde{P'_n}, \tilde{i'})$ The content modeled by R' will be removed from the processed instance of P'_o. Since in the new version, $\widetilde{R'}$ is among contents of $\widetilde{P'_n}$, the wrapping XML subelement is created (if $\widetilde{R'}$ has a name) in the instance of $\widetilde{P'_n}$.

If $getInVer(child(\widetilde{R'}), v) = child(R')$, i.e., the association was moved with its child (which is the usual situation, but not the general case), the adaptation proceeds to the child, i.e., the instances of $child(R')$ will be converted to instances of $child(\widetilde{R'})$. The situation is similar to adaptation of *attributeMoved* change described in Section 4.2.2. The adaptation is again largely affected by the cardinalities of the concerned association and the positions of the nodes P'_o and $\widetilde{P'_n}$. In case of $getInVer(\widetilde{P'_n}, v) = P'_n \neq \perp$ and $stable(tree(P'_o, P'_n))$, we can distinguish

some cases corresponding to those proposed for *attributeMoved* and offer similar options for adaptation. Otherwise, the user must provide his/her adaptation function for the particular case.

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

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

classMoved **($\widetilde{C'}$, $\widetilde{R'_n}$), *contentModelMoved* ($\widetilde{M'}$, $\widetilde{R'_n}$)** As opposed to the previous case, it is now the node which is moved, not the association leading to it. We introduce separate predicates to cover all possible changes between two version of the model, but their treatment during adaptation is analogous. Let $\widetilde{N'} = \widetilde{C'}/\widetilde{M'}, N' = getInVer(\widetilde{C'}, v)/getInVer(\widetilde{M'}, v)$ for $classMoved/contentModelMoved$ respectively. The instances of N' will be converted to the instances of $\widetilde{N'}$ in the content of $\widetilde{R'_n}$. Again as for *attributeMoved* and *associationMoved*, adaptation can be offered in some particular cases, but for the general case, the user must provide his/her adaptation function.

In the rest of this section, we will first discuss approaches in situations, where the adaptation algorithm should create new content in the adapted document. In the end of this section, we show a larger example of schema evolution and document adaptation.

4.2.6 Generating content

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

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

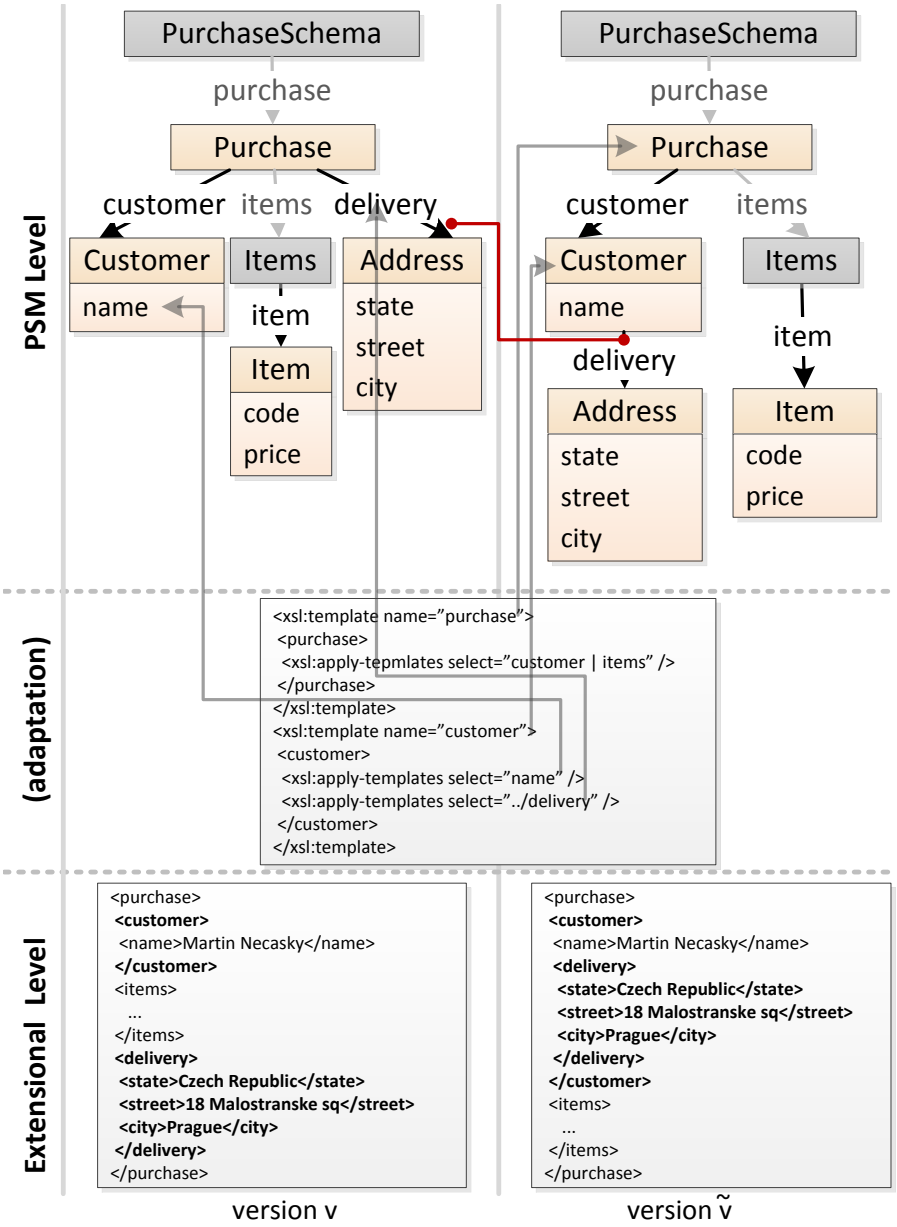


Figure 4.2: associationMoved – adaptation

The following text discusses several possible solutions.

Default values of attributes One of the easiest ways is to introduce function *default* that would assign a default value for PSM attributes. This value could then be used each time an attribute instance needs to be generated. XML schema languages routinely provide constructs for specifying default values of attributes, so the result is always defined. However, such a solution can not be used when no default value can be found for an attribute (e.g., when the value should be computed based on other data in the document).

Default complex content The adaptation script can also create the missing element content itself. The structure of the internal nodes is given, for values of leaf nodes and attributes the default values for the given type of each PSM

attribute (e.g., an empty string for type `xs:string`) can be utilized or the attributes can filled with default values specified by the user, as suggested in the previous paragraph. Where a content model of type choice is present, the first component is always selected. For each attribute and association with cardinality $m..n$, the algorithm creates exactly m instances. Such a content will be called *default instance*. The default instance can serve as a skeleton for the user, who can then further modify it, either in the adaptation script or, after adaptation, in each adapted document.

Advanced techniques The previous techniques result in the same adaptation for every document and they cannot handle situations, where adaptation script is supposed to create different content for every document. In Chpt. 6, we show how to derive new content from the content of the adapted document. In Chpt. 9.3, we discuss the possibilities of creating content based on some external data sources.

4.2.7 Adaptation Example

To demonstrate changes and the adaptation script, we provide an example of schema adaptation for a purchase schema in Fig. 4.3, XSDs for the PSM schemas can be found in Appendix A.3 – A.4. The figure depicts the version links for the changed constructs (the links for the unchanged constructs are hidden from the figure). All constructs participating in changes are also highlighted.

In the new version, association `address` was moved from `CustomerInfo` to `Customer` and renamed to `delivery-address`. New classes `Items` and `CustEmail` were added. Attribute `email` was moved from `Customer` to `CustEmail` and its cardinality was restricted to 0..5. Attributes of `Address` class were reordered and attribute `weight` was removed from the schema. The change instances as detected by Alg. 1 are as follows (we use subscripts 1 and 2 to distinguish constructs from $\mathcal{S}'$ and $\widetilde{\mathcal{S}}'$):

- *associationMoved*(`delivery-address`$_2$, `Customer`$_2$, i)
- *associationEndRenamed*(`delivery-addres`$_2$, *'delivery-address'*)
- *classAdded*(`Items`$_2$, `Purchase`$_2$)
- *associationMoved*(`item`$_2$, `Items`$_2$, 1)
- *classAdded*(`CustEmail`$_2$, `Customer`$_2$, 2)
- *attributeMoved*(`email`$_2$, `CustEmail`$_2$, 1)
- *attributeCardinalityChanged*(`email`$_2$, 0..5)
- *attributeRemoved*(`weight`$_1$)
- *attributeIndexChanged*(`city`$_2$, 1)
- *attributeIndexChanged*(`street`$_2$, 2)
- *attributeIndexChanged*(`zip`$_2$, 3)

In the next section, we will describe how changes are adapted in our framework and an adaptation script is generated as an XSL transformation.

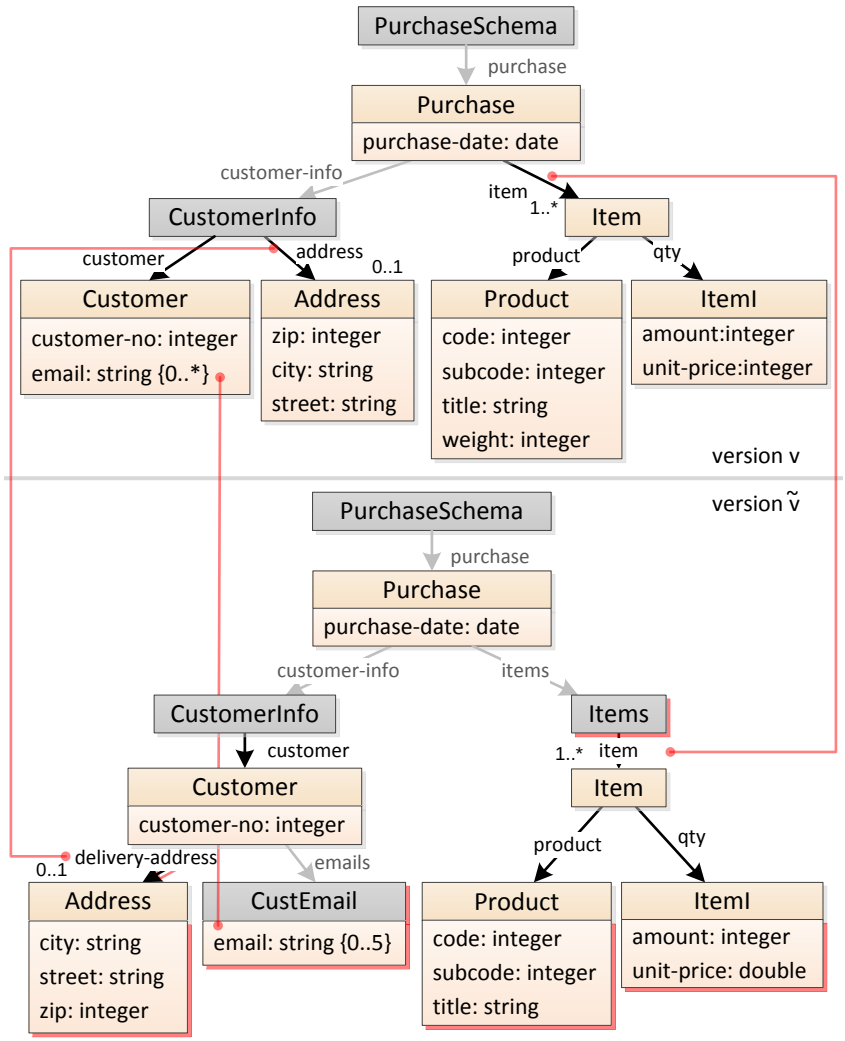


Figure 4.3: Adaptation example: schemas

4.3 Implementation in XSLT

In this section, we describe how our framework creates the adaptation script using XSL.

The change detection algorithm (Alg. 1) outputs the set of changes between the schemas. Having the set of changes, we can now describe the algorithm for producing an adaptation script that outputs document $\widetilde{D'}$ adapted to $\widetilde{\mathcal{S}'}$ when applied on XML document D' valid against $\mathcal{S}'$.

The sequence of changes made over a PSM schema can be converted to a script/expression in one of the languages for querying and/or manipulating XML, be it *XQuery* [6], *XQuery Update Facility* [91], XSL [87] or DOM [81].

Assuming that the *XQuery Update Facility* is the implementation language, each change 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 adapted document. The procedure when using *DOM API* would be analogous in that the document is updated in place.

Our approach uses XSL stylesheets as the implementation language due to the wide support for XSL among the tools working with XML data and especially the database systems supporting XML Schema [92] evolution.

The following properties of XSL as a language must be addressed when designing the adaptation algorithm:

1. *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. This is achieved either by designing the script in such a way 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.
2. *Outputting of unchanged content*: In XSL, everything that should appear in the output must be sent there (using XSLT instructions) explicitly. When some part of the document was not affected by any changes, there still must be an instruction that outputs the unchanged part to the result.
3. *Output definitiveness*: When XSLT processor sends a content to the output, it can not be changed during the same transformation[1], the changes have to be grouped and conducted together.

Due to space limitations, we will show how the algorithm processes only the core constructs (classes, attributes with $xform = e$ and associations with $name \neq \lambda$). According to the categories of changes, the first step is to divide the PSM constructs the schema into disjoint sets $\mathcal{K}_a$, $\mathcal{K}_r$, $\mathcal{K}_m$ and $\mathcal{K}_s$ (of added, removed, moved and sedentary constructs) and also classifies nodes $\widetilde{\mathcal{N}'} \cup \widetilde{\mathcal{S}'_a}$ (i.e., we treat attributes as nodes in the algorithm) in schema $\widetilde{\mathcal{S}'}$ the tree into three disjoint groups:

– *red nodes* – the nodes in $\widetilde{\mathcal{S}'}$ that were changed or added + old and new parent nodes of all the migrated and renamed nodes/associations + classes that contain changed attributes + classes from which attributes were moved or removed

– *blue nodes* – nodes that are not red, but contain a red node in their subtrees

– *green nodes* – other nodes

The motivation for this division is: red nodes need to be adapted explicitly; blue nodes do not need adaptation themselves, but the processor must descend into their subtree (because there is at least one red node there); green nodes do not need adaptation and their subtree by definitions contains only other green nodes – so they can be copied to the result as they are, with the whole subtree, without

[1]Unless several transformations are pipelined, but we want our adaptation script to be one-pass.

further processing.

In XSL, stylesheets producing the same output can be written in several forms. To keep it transparent, comprehensible and easily modifiable, the generated adaptation stylesheet $\mathcal{F}$ takes the following form:

- It is a one-pass stylesheet.
- It follows the *navigational stylesheet* pattern described in [32]. It relies on a detailed knowledge of the input document. XPath expressions used for `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 templates 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 schema, not with the complexity of the schema.

The main steps of the adaptation algorithm are depicted as Alg. 2. The algorithm creates $\mathcal{F}$, the adaptation script for $\mathcal{S}' \to \widetilde{\mathcal{S}}'$ document adaptation in the form of an XSL stylesheet. In the rest of this section, we will describe how the steps on lines 5, 7 and 8 are realized, i.e., how are the templates for red, blue and green nodes created.

Algorithm 2 GenXSLT

Input: old and new version $v, \tilde{v} \in \mathcal{V}$, PSM schemas $\mathcal{S}'$, $\widetilde{\mathcal{S}}'$
Output: $\mathcal{F}$, an XSLT stylesheet for $\mathcal{S}' \to \widetilde{\mathcal{S}}'$ adaptation
1: $C_{\mathcal{S}',\widetilde{\mathcal{S}}',v,\tilde{v}} \leftarrow$ result of Alg. 1 *DetectChanges* for $v, \tilde{v}, \mathcal{S}', \widetilde{\mathcal{S}}'$
2: label nodes $\widetilde{\mathcal{N}'} \cup \widetilde{\mathcal{S}'_a}$ *red*, *blue* and *green* acc. to $C_{\mathcal{S}',\widetilde{\mathcal{S}}',v,\tilde{v}}$
3: Distribute $\widetilde{\mathcal{N}'} \cup \widetilde{\mathcal{S}'_a}$ into groups $\mathcal{K}_a$, $\mathcal{K}_r$, $\mathcal{K}_m$ and $\mathcal{K}_s$ acc, to $C_{\mathcal{S}',\widetilde{\mathcal{S}}',v,\tilde{v}}$
4: **for all** $\widetilde{N'} \in$ *red nodes* **do**
5: generate a top level template $T_{\widetilde{N'}}$
6: **end for**
7: generate T_B – a template processing all *blue nodes*
8: generate T_G – a template processing all *green nodes*

We start by showing the templates that process the blue and green nodes and then we show how templates processing the red nodes are constructed. For processing blue and green nodes, $\mathcal{F}$ contains templates depicted in Figure 4.4. The first template copies an element with its attributes and instructs the processor to continue with its subelements (where at least one element corresponding to a red node exists, which must be revalidated). The second template copies the element with its whole subtree to the output. Since all red and green nodes are

processed by these two templates, the complexity of $\mathcal{F}$ does not grow with the size of the schemas, but with the amount of changes mad between $\mathcal{S}'$ and $\widetilde{\mathcal{S}}'$.

For each red node $\widetilde{N'}$ the algorithm generates one template in $\mathcal{F}$. During the process, the algorithm keeps a track of the currently processed node in the source schema (available through a variable `processedPath`). The algorithm can compute the XPath expression that selects instances of a given node in the input document from the processed node via function $relativeXPath(X', processedPath)$. Here are some example results of *relativeXPath* for Figure 4.3:

`processedPath`	**Path to node** X'	**Relative path**
`/Purchase`	`/Purchase/Item/@amount`	`Item/@amount`
`/customer-info/Customer`	`/customer-info/Address/city`	`../Address/city`

Figure 4.5 shows the basic structure of the template. It uses several auxiliary functions. For each instance of class C', we will denote $inAssociation(\widetilde{C'})$ the association $\widetilde{R'}$ s.t. $child(\widetilde{R'}) = \widetilde{C'}$ and $\widetilde{R'}$ was used when recognizing the instance (there may be more associations leading to a class). Auxiliary function *elementName* returns the $name(\widetilde{A'})$ for attribute $\widetilde{A'}$ and $name(inAssociation(\widetilde{C'}))$ for class $\widetilde{C'}$. Function `suggestName` returns a unique, but human-friendly name for the red node template. Finally, $childNodes(\widetilde{N'})$ are simple the children of $childAssociations(\widetilde{N'})$.

If the processed node is an added node (i.e., $\widetilde{N'} \in \mathcal{K}_a$), it will be a named template (an auxiliary). Otherwise, it will be a template with `match` attribute.

The template creates the literal element corresponding to the node (using *elementName*), then `processConstruct` subroutine is called for processing each attribute (if $\widetilde{N'}$ is a class and not a content model) and the same subroutine is called also for each child of the processed node.

If the node is an attribute, `xsl:value-of` is used to retrieve its value and copy it to the result. If the type of the attribute changed (*attributeTypeChanged* is detected, in that case let D' be the old type of the attribute and $\widetilde{D'}$ the new one), a conversion function must be called. In the pseudocode, this call is represented by the function $conv_{\widetilde{N'}} : domain(D') \rightarrow domain(\widetilde{D'})$. If the type did not change, the call can be omitted ($conv_{\widetilde{N'}} = identity$), similarly in the case when $domain(D') \subseteq domain(\widetilde{D'})$ (which is guaranteed e.g., when D' is a subtype of $\widetilde{D'}$, see adaptation of *attributeTypeChanged* in Sec. 4.2.2 earlier in this chapter). Subroutine `processConstruct` examines the state (whether the construct belongs to the set of added nodes) and also its cardinality.

```
<xsl:template match="{blue-nodes-paths}">
  <xsl:copy>
    <xsl:copy-of select='@*' />
    <xsl:apply-templates select='*' />
  </xsl:copy>
</xsl:template>
<xsl:template match='{green-nodes-paths}'>
  <xsl:copy-of select='.' />
</xsl:template>
```

Figure 4.4: Green and blue nodes template

```
<xsl:template
  { if N'~ ∉ K_a } match='{N'~.XPath}' {else} name='{suggestName(N'~)}'>
  <{elementName(N'~)}>
    { if N'~ ∈ S'_c then foreach Ã ∈ attributes(N'~)
        processConstruct(Ã, card(Ã))
      if N'~ ∉ S'_a foreach C̃ ∈ childNodes(N'~)
        processConstruct(C̃, card(inAssociation(C̃))) }
    { else } // attribute(leaf) → add the value
        <xsl:value-of select='{conv_N'~}{(relativeXPath(N'~, processedPath))}' />
  <{elementName(N'~)}>
</xsl:template>

procedure processConstruct
  parameter N'~ ∈ S'_a ∪ S'_c // processed attribute or class
  parameter l..u // cardinality
{
  case N'~ ∈ K_a ∧ l = 0 :
    exit; // added optional element can be skipped
  case N'~ ∈ K_a ∧ l > 0 :
  case N'~ ∈ K_s ∪ K_m ∧ cardinalityChanged(N'~):
    generateElementCardinalityReference(N'~, l..u)
  otherwise: // added with card = 1 or cardinality unchanged
    generateElementSingleReference(N'~)
}
```

Figure 4.5: Red nodes template – structure

Function *cardinalityChanged* looks up *associationEndCardinalityChanged*/*attributeCardinalityChanged* change (if there is one). There are two variants of *reference generating* subroutine – *single* (not dealing with cardinalities) and *cardinality* (designed to adapt to changes in cardinality). The first one is depicted in the first part of Figure 4.6.

If the processed node is added, call of *instance generator* template for the node is added to $\mathcal{F}$. The *instance generator* template $instanceGenerator_{\widetilde{N'}}$ creates the *default instance*, as described earlier in this chapter in Sec. 4.2.6 (to be more prices, it creates the n instances, where n is the value passed to the `count` parameter). The template $instanceGenerator_{\widetilde{N'}}$ can be modified by the user after the script is generated. In Chpt. 6 we will also provide an alternative how the generated instance can be created based on the content of the adapted document.

If the process node is not added, `xsl:apply-templates` is outputted (with possible *condition* – a parameter that is used when the *single* variant is called from the *cardinality* variant.

Finally, the *cardinality* variant of reference generating is depicted in the second part of Listing 4.6. There are two parts of the template. The first part concentrates on instances already present in the document (and is therefore skipped for *added* elements). Existing instances are processed again by the *single reference* subroutine – either all existing instances (when the upper cardinality of node $\widetilde{N'}$ 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* variant 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: either $\widetilde{N'}$ is an node with lower cardinality > 1 or the lower cardinality of $\widetilde{N'}$ was increased. Again, `instanceGenerator` template is made responsible for creating new instances.

In Sec. 4.1, we identified several changes where more than one option to adapt

procedure generateElementSingleReference
  parameter: $\widetilde{N'} \in \widetilde{\mathcal{N}'} \cup \widetilde{\mathcal{S}'_a}$ // *referenced node*
  parameter: condition: XPath expression **optional**
{ **if** $\widetilde{N'} \in \mathcal{K}_a$ }
  <**xsl:call−template name**='{$suggestName(\widetilde{N'})$}' />
{ **else** }
  { **var** xpath ← $relativeXPath(\widetilde{N'}, processedPath)$ }
  { **if** condition **is set** }
    <**xsl:apply−templates select**='{xpath}[{condition}]' />
  { **else** }
    <**xsl:apply−templates select**='{xpath}' />
  { **end if** }
{ **end if** }

procedure generateElementCardinalityReference
  parameter: $\widetilde{N'} \in \widetilde{\mathcal{N}'} \cup \widetilde{\mathcal{S}'_a}$ // *referenced node*
  parameter $l..u$ // *cardinality*
  /* *routine called either when cardinality of element N changed or N' was added with lower cardinality > 1* */
{ **if** $\widetilde{N'} \in \mathcal{K}_s \cup \mathcal{K}_m$ // *existing node*
  // *cardinality of N' changed, deal with existing nodes*
  if $\neg u$ *decreased from prev. version*
    generateElementSingleReference($\widetilde{N'}$)
  else
    generateElementSingleReference($\widetilde{N'}$, condition = '$position() \leq$ '.u)
  end if
end if
if $\widetilde{N'} \in \mathcal{K}_a \vee$ *lower cardinality of* $\widetilde{N'}$ *increased*
  // *new nodes need to be created*
  var countExpr
  var lower ← $l(card)$
  if ($N' \in \mathcal{K}_a$)
    countExpr ← lower
  else
    var existing ← $relativeXPath(\widetilde{N'}, processedPath)$
    var countExpr ← l .' − **count**('. existing .')'
    <**xsl:call−template name**='{$instanceGenerator_{\widetilde{N'}}$}'>
      <**xsl:with−param name**='count' **select**='{countExpr}' />
    </**xsl:call−template**>
  { **end if** }
{ **end if** }

Figure 4.6: Generating element reference

a certain change exists. Our algorithm described in this section is straightforward in those cases – where instances need to be created for cardinalities $m..n$, m cardinalities are created, for *choice* content model, the first choice is always selected etc. In Chpt. 6, we show how this behaviour can be refined using expressions/ annotations.

To conclude this chapter, we refer to the example of an adaptation stylesheet generated for the scenario from Fig. 4.3. The full text of the stylesheet can be found in Appendix B.1.

5. Expressions in the Model, Integrity Constraints

In this chapter, we will discuss the expression languages used at different levels in our framework. Results presented in this chapter were published in [47, 31].

Structural diagrams are a very useful tool when describing a software system. However, there are always limits in their expressive power. It is a common practice to describe the structure using a diagram and provide additional detailed information about integrity constraints (invariants, pre- and post-conditions) or some other, more fine-grained, information about the system in a textual form. If a formal language is used instead of the natural language, the information is guaranteed to be unambiguous and accurate.

Formal languages for these purposes are usually some kind of expression/ functional languages. An expression language can be used not only to describe a system, but also to define actual queries which the system supports and which can be executed at runtime. We will use this property so that our framework allows not only, e.g., define the integrity constraints that the system must meet, but also to generate code that actually verifies the integrity constraints in a running system.

In the following three sections, we will describe a suitable expression language for the PIM, PSM and operational level. For the PIM and operational levels, we will use existing, well-estabilished languages (OCL [66] and XPath [89]). For the PSM level, we will use the PIM level language with certain modifications. The main topic of this chapter is the automatic translation of expressions from the PIM to the PSM and operational level. This will allow us to evaluate expressions and verify integrity constraints, which are defined at the abstract level, in XML documents, without the need to manually adjust each expression for a concrete document structure (schema).

We will refer to the PIM schema in Fig. 5.1 and the PSM schemas in Fig. 5.2 and 5.3. Translations of these schemas into XSDs can be found in Appendix A.5, A.6 and A.7 respectively. We use several PSM schemas in this chapter to demonstrate how a PIM expression is translated into different PSM expressions in different PSM schemas.

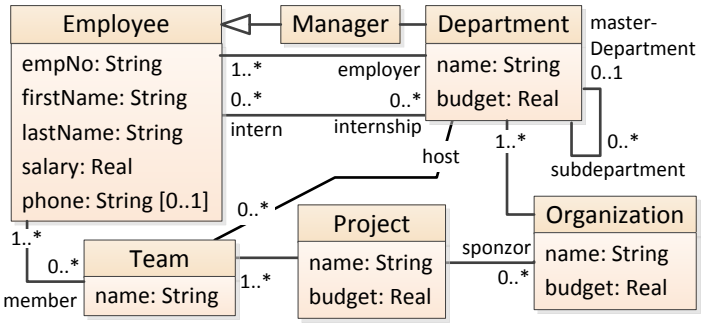


Figure 5.1: Sample PIM schema – organization

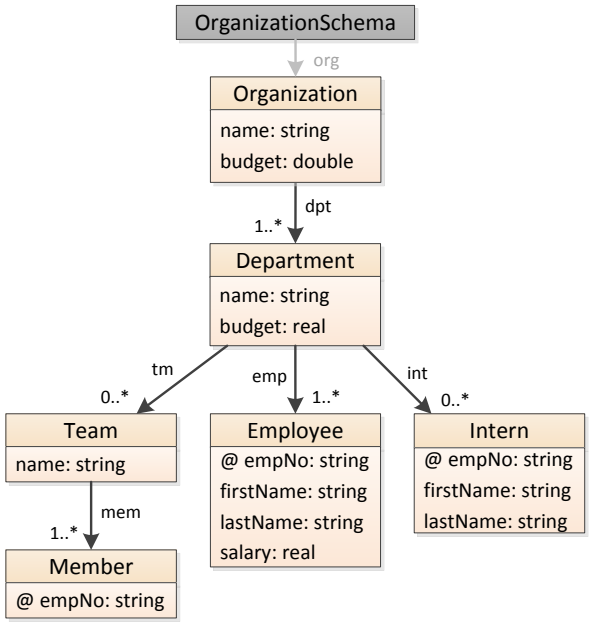


Figure 5.2: PSM schema showing distribution of employees in departments

5.1 OCL Expressions at the PIM Level

Since our PIM schemas are UML class diagrams, we can directly use the Object Constraint Language (OCL [66]), which is a part of the UML standard, at the PIM level.

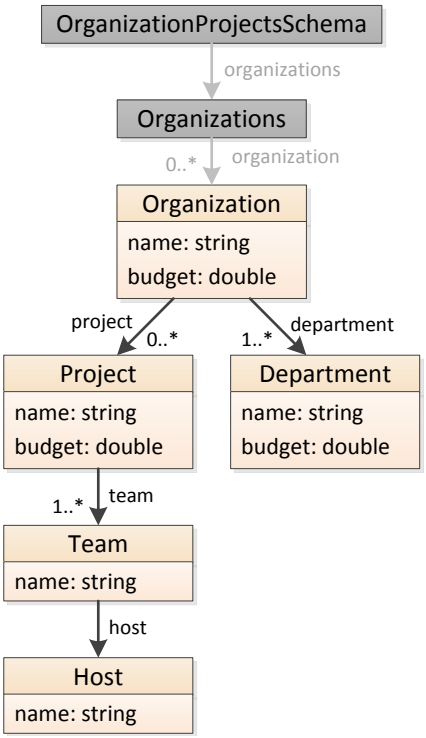
OCL is a text based language, combining mathematical notation (used in, e.g., first-order logic expressions) and principles known from functional languages with object-oriented model. Its grammar allows for recursive building of formulas from subformulas. The syntax and semantics of OCL is formally introduced in [74]. The meta-model of the core of OCL is depicted in Figure 5.4a. The figure illustrates how expressions are composed of subexpressions. In the rest of this article, OCL expressions in the text will be delimited using guillemots, e.g., this: «$x + y > 1$» is an OCL expression. We will use large uppercase letters to denote OCL expressions at the PIM level (usually O). An apostrophe will be used to denote OCL expressions at the PSM level (e.g., O'). The following example shows how OCL expressions can be used in invariants:

context Organization **inv** PIM_IC1:
self.Department.Team->forAll(t |
t.member->size() $<$ 0.1 $*$ self.Department.employee->size())

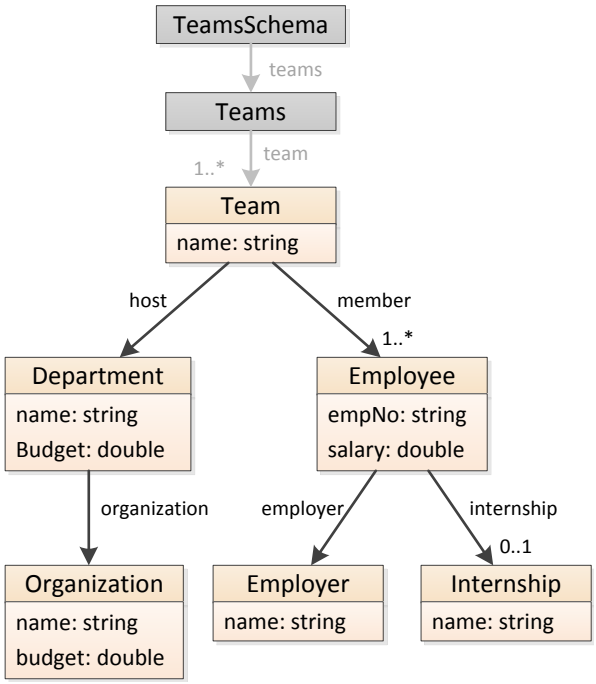
The constraint specifies that in an organization, each team in any department cannot have more than 10% of the total number of employees of the organization.

The first line specifies the *context* (class `Organization`) and that what follows is an invariant[1] (with identifier PIM_IC1). Every invariant contains an expression which is evaluated at the instance level to *true* or *false*. For a model to be valid at a given time, all invariants must evaluate for all instances to *true*. Each subexpression in the invariant PIM_IC1 is an instance of some class from the metamodel in Fig. 5.4a (details of `FeatureCallExp` are depicted in a separate

[1]Later in this chapter we also describe different usages of OCL, besides invariants.



(a) A PSM schema modeling a type of XML documents with organizations which contain projects and departments in the organization. A project contains teams and teams contain hosting departments.



(b) A PSM schema modeling a type of XML documents with teams. A team contains a hosting department and organization. It also contains its members. For each member, there is its employing department and, optionally, a department where the member is currently doing his or her internship.

Figure 5.3: PSM schemas focusing on teams and projects

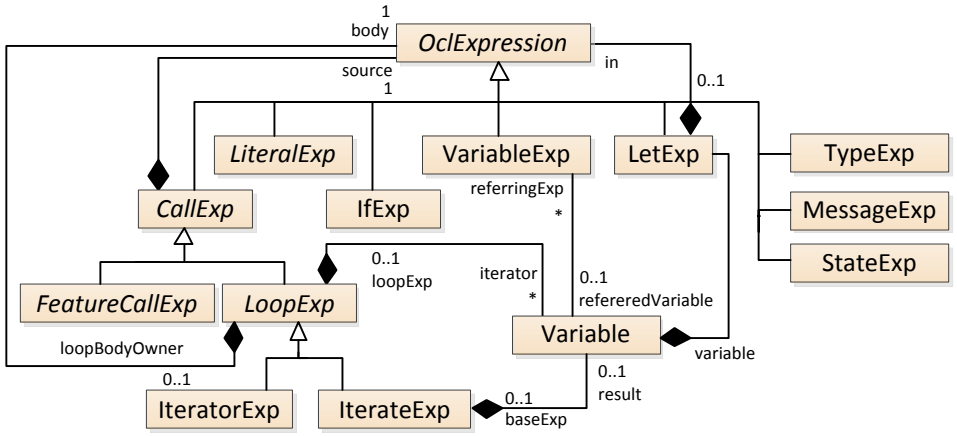
Fig. 5.4b), e.g.,:

- «*self*» is an instance of `VariableExp` referring to variable *self* (context variable),
- «*self.Department*» is an instance of `PropertyCallExp` where «*self*» is the `source` and «*Department*» the `referredProperty`,
- «$forAll(t \mid t.member\ldots)$» is an instance of `IteratorExp` with `iterator` t and the `body` subexpression is the expression starting with «*t.member*»,
- «$0.1 * self.Department.employee\texttt{->}size()$» is an example of the metaclass `OperationCallExp` with two `arguments` and the `referredOperation` being multiplication ($*$).

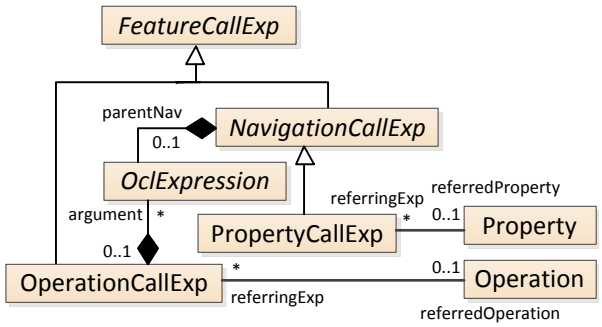
Invariants are one of several possible uses of OCL expression. OCL specification also defines how to use OCL expressions to

- define initial values of properties (attributes and association ends),
- define pre-conditions and post-conditions for methods,
- add new properties and operations in pure OCL.

In this chapter, we will show how OCL expressions can be translated into XPath expressions and how they can be used to validate XML documents using Schematron, and how to annotate version links to override the default behavior of the document adaptation algorithm presented in Chpt. 4.



(a) OCL expressions metamodel – overview



(b) Detail of `FeatureCallExp`

Figure 5.4: Different kinds of OCL expressions (source: OCL specification, Chapter 8.3)

5.2 OCL Expressions at the PSM Level

Since we are using modified class diagrams at the PSM level, we can also use OCL to write expressions over PSM schemas. It would be possible to define translation of PIM OCL expression directly to the XML languages of the operational level (see the next Sec. 5.3), but we decided to use OCL at the PSM level as well – OCL is better suited for class diagrams (even at the PSM level), it can be managed more easily when schemas change and transformation of constraints from the PIM level is more transparent when the same language is used. Last, our framework has strong support for evolution of schemas based on change propagation [56]. We leave the evolution of expression/queries to the future work (see Chpt. 9), but integrating PIM OCL $\leftrightarrow$ PSM OCL change propagation into the algorithm will be easier than propagating changes between PIM OCL and expressions in XML query languages.

However, since we modified class diagrams to fit the needs of XML modeling, we will extend OCL language for the PSM level as well. This section describes the extensions of OCL for the PSM level.

Traversing among related PSM classes Several PSM classes can be mapped to the same PIM class *interpretation* (which is the case of classes `Intern′` and `Employee′` in Fig. 5.2, both representing PIM class `Employee`). This property of PSM schemas requires special support in our PSM OCL expressions. Let

us consider the following PIM OCL expression specifying an invariant ensuring that interns are making their internships in other departments than their home department.

context e:Employee **inv** PIM IC3:
e.internship <> e.employer

Suppose that we want to transform this PIM OCL expression to a PSM OCL expression expressed over the PSM schema depicted in Fig. 5.2. We must act cautiously because for class `Intern′` there is no association end mapped to the end `employer` in the PIM schema and for class `Employee′`, there is no association end mapped to `internship`. However, PSM classes `Employee′` and `Intern′` are linked semantically (they have the same interpretation – class `Employee` in the PIM schema). Therefore, a single instance of class `Employee` may be represented by an instance of `Employee′` and a different instance of `Intern′` in the same XML document. Therefore, no straightforward variant of the OCL expression above will work for the PSM schema.

For this purpose, we introduce a new function that allows for traversing a PSM schema along these semantic links. In fact, we introduce a function for each class C' in the PSM schema. The name of the function is *to* followed by the name of C'. The function has no parameters. For example, for class `Employee′` has name *Emloyee*. Therefore, we introduce a function *toEmployee*().

The function can be called on a source expression of type C'_1 s.t. interpretation of both C' and C'_1 is the same PIM class C. It returns a set of all instances of C' which represent the same instance of C as the instance of C represented by C'.

context i: Intern **inv** PSM_IC3:
i.Department <> i.toEmployee().Department

Evaluation of the expression starts in the PSM class `Intern′` and needs to evaluate two navigation paths. The former path needs to navigate from an instance i' of `Intern′` to an associated instance di' of `Department′` via an association end which represents the end `internship` from the PIM schema. This is possible because the PSM association having `Intern′` as its child represents the PIM association with `internship` as an end. However, it also needs to navigate from i' to another associated instance de' of `Deparment′` via an association end which represents the association end `employer` from the PIM schema. This is not possible. The OCL expression has to traverse from the `Intern′` instance to the corresponding `Employee′` instance using the function *toEmployee′*. From here, the required navigation is possible.

In the previous sample, the two navigation subexpressions («*i.Department*» and «*i.toEmployee().Department*») navigate the model in the upwards direction. Both refer to the parent association end using the name of the class. Another option would be to use the name of the association end, which is always 'parent' in the case of parent association ends. The previous PSM OCL constraint can be alternatively expressed as follows:

context i: Intern **inv** PSM_IC3:
i.parent <> i.toEmployee().parent

Let us note that at the PIM and PSM levels, which are conceptual, we do not solve how the function is implemented. We consider an object identity of the instances which allow the traversing. In the phase of translation of PSM OCL

constraints to XPath (which is the subject of Sec 5.5), we show how we deal with the added functions at the XPath level (we use PIM keys).

Navigating to content models In Definition 2, we introduce a PSM-level specific construct – the content model. Content models behave like classes without attributes – they participate in associations. However, they do not have a name and the child end of their parent association is also unnamed in normalized schemas. To allow OCL navigations, we need a way to identify content models in the expressions. For this purpose, we introduce a variant of *PropertyCallExp* relying on the position of the association among the child associations of a node.

We allow to expression such as «*c.choice*1», «*c.set*2» and «*c.sequence*3». A *PropertyCallExp* written as «*c.cmtypeN*» where c is of type C' looks-up the association end leading to the N-th content model of type *cmtype* (choice, set or sequence) among the child association ends of the associations in *childAssociations*(C').

5.3 Expressions at the Operational Level – XML Queries

XML has an estabilished expression language – XPath, which is one of the most fundamental technologies of the XML stack. The current recommendation is 2.0 [89], version 3.0 [94] is now a candidate recommendation.

XPath can be used to select nodes in an existing documents and work with atomic values. XPath does not provide capabilities to create nodes not present in the queried document neither to define user functions[2]. XPath 2.0 data model (shared by XSLT and XQuery) knows only 1 kind of collection – a sequence. XPath 3.0 adds another kind – a map (associative dictionary).

XPath is used by many other languages in the XML stack, including XQuery [6] (which is a superset of XPath) and XSLT [87].

We will use XPath as the ground expression language at the operational level. In cases where we need to extend it, we will use XSLT, but XQuery could be used as well.

The rest of this chapter presents one of the major contributions of this thesis – translating expressions from the PIM to the PSM and operational level. Using our results, it is possible to define queries and constraints at the abstract platform-independent level and let the framework decide in which PSM schemas the constraints apply. In the next step, the framework can automatically generate operational-level code to verify the integrity constraints or execute the queries.

5.4 From PIM Level Expressions to PSM Level Expressions (PIM OCL → PSM OCL)

In this section, we will discuss translation of PIM expressions into PSM expressions.

[2]XPath 3.0 adds the possibility to define and call anonymous functions

When we want to tranlsate a certain PIM constraint O into a PSM constraint O', a large part of the translation is straightforward – e.g., a `+` (integer addition operation) or `'word'` (a string literal) stay the same. What we have to focus on are only those parts of expressions that involve the constructs of the model – classes, associations and attributes. Thus, for in this section, we will suffice only with a subset of OCL and formalize only those parts of OCL expressions which are important for the translation between PIM and PSM levels. We introduce *navigation paths*, e.g., «*self.Department.Team*». A navigation path starts in a variable and navigates via one or more steps in the structure of the PIM schema.

At the instance level, the evaluation of such navigation path starts in an instance c of a class which is assigned to the variable and navigates to a collection of instances associated with c via the navigation path. A path may be followed by a *collection function*, e.g., *forAll* or *size*, which is a kind of iteration function. It iterates the collection resulting from the evaluation of the path. It may return another collection or a simple data value. Formally, a step is also a collection function which iterates the result of the previous step, evaluates the specified navigation for each iteration and returns the new collection or value. The syntax of OCL allows also for navigation paths which do not start in a variable, but, e.g., at a collection function. However, we do not consider this kind of navigation paths in this thesis for simplicity. Let us only note that these paths can be translated to those which start in a variable.

Having an OCL expression O over a PIM schema $\mathcal{S}$, we only translate its context class and navigation paths. The other parts of O remain untouched. Therefore, we will use a simplified formal model which represents O as a pair $(C, \{p_1, \ldots, p_n\})$ where C is the context class C and each p_i is a navigation path in $\mathcal{S}$. A *navigation path* p in $\mathcal{S}$ is a construction s_0 . ... . s_n where each s_i is called *step* of p. The first step s_0 is a variable. We distinguish the context variable (denoted *self*) and the other user defined variables (usually denoted by a character, e.g., v, k, etc.). A step s_i specifies an attribute A (then s_i is the name of A) or association endpoint E (then s_i is the name of E or the name of the association E belongs to). If s_i is an attribute then $i = n$.

Our sample OCL expression above is represented in our formalism as a pair:

(*Organization*,
{«*self.Department.Team*», «*t.member*», «*self.Deparment.employee*» })

In this section, we show how PIM OCL expressions are translated to their PSM OCL equivalents. For this section let $\mathcal{S}$ be a PIM schema and $\mathcal{S}'_1, \ldots, \mathcal{S}'_n$ be PSM schemas, each with an interpretation function I_i against $\mathcal{S}$. Our goal is to translate a PIM OCL expression O over $\mathcal{S}$ to O'_i for each $\mathcal{S}'_i$ whenever it is meaningful. The resulting O'_i must specify the same constraint over $\mathcal{S}'_i$ as specified by O over $\mathcal{S}$. Let us note that a PSM schema often does not contain representations of the whole problem domain described in the PIM (a PSM schema may contain several classes relevant for a certain component of the system, while others, which are not important for the component, are not represented in the schema). It is clear that it is meaningful to translate O only when all classes, attributes and associations in $\mathcal{S}$ restricted by O are represented in $\mathcal{S}'_i$. In that case, we will say that $\mathcal{S}'_i$ *covers* O. Formally:

Definition 13. *Let $\mathcal{X}$ be the set of all PIM constructs referenced from the PIM*

Algorithm 3 Translating a PIM OCL navigation path to a PSM equivalent

Input: PIM OCL navigation path $p = s_0, \ldots, s_m$ in a PIM OCL constraint O
Output: PSM OCL navigation path p' translated from p in a PSM OCL constraint O' translated from O

1: $p' := s_0$
2: **if** $s_0 =$ `self` **then**
3: $\quad C'_1 :=$ context class of O'
4: **else**
5: $\quad C'_1 :=$ `target`$(\mathit{superior\text{-}path}(p'))$
6: **end if**
7: **for** $i \in 1 \ldots m$ **do**
8: $\quad C_i := I(C'_i)$
9: $\quad X_i :=$ `target`(s_i, C_i)
10: $\quad$ **if** $X_i \in \mathcal{S}_a$ **then**
11: $\quad\quad$ Let $X'_i \in \mathcal{S}'_a$ s.t. $\mathit{class}(X'_i) = C'_i$ and $I(X'_i) = X_i$
12: $\quad\quad p' := p'$. $\{\mathit{name}(X'_i)\}$
13: $\quad$ **else**
14: $\quad\quad$ Let $R_i \in \mathcal{S}_r$ and $Y_i \in \mathcal{S}_e$ s.t. $R_i = \{X_i, Y_i\}$
15: $\quad\quad$ Let $X'_i \in \mathcal{S}'_r$ s.t. $I(X'_i) = R_i$ and $\mathit{parent}(X'_i) = C'_i$ or $\mathit{child}(X'_i) = C'_i$
16: $\quad\quad$ **if** $\mathit{parent}(X'_i) = C'_i$ **then**
17: $\quad\quad\quad p' := p'$. $\{\mathit{name}(X'_i) = \lambda\ ?\ \mathit{name}(X'_i) : \mathit{name}(\mathit{child}(X'_i))\}$
18: $\quad\quad\quad C'_{i+1} = \mathit{child}(X'_i)$
19: $\quad\quad$ **else**
20: $\quad\quad\quad p' := p'$. `parent`
21: $\quad\quad\quad C'_{i+1} = \mathit{parent}(X'_i)$
22: $\quad\quad$ **end if**
23: $\quad\quad$ **if** $i = m$ **then**
24: $\quad\quad\quad$ `target`$(p') := \mathit{participant}(Y_i)$
25: $\quad\quad$ **end if**
26: $\quad$ **end if**
27: **end for**
28: **return** p'

constraint O. We will say that PSM schema $\mathcal{S}'$ covers O iff $\forall X \in \mathcal{X}\ \exists X' \in \mathcal{S}'_c \cup \mathcal{S}'_a \cup \mathcal{S}'_r \cup \mathcal{S}'_e$ s.t. $I(X') = X$.

5.4.1 Direct Translation

We will first discuss a *direct translation* of the PIM OCL expression O to its equivalent PSM OCL expression O' over a given PSM schema $\mathcal{S}'$. Basically, the direct translation starts with replacing the context class C of O with its representation C' in $\mathcal{S}'$ (i.e., $I(C') = C$). C' becomes the context class of O'.

The translation then proceeds with translating all navigation paths in O. The paths of O form a hierarchy. Each path p consisting of steps $s_0, \ldots, s_m$ is a node of this hierarchy. The first step s_0 is always a variable. When $s_0 =$ *self*, then p is at the top-level of the hierarchy. Otherwise, s_0 is a variable v declared by a collection function which follows another navigation path $\bar{p}$ in O,

i.e., «$\bar{p}$->*colf*(v | Q)», where Q is a PIM OCL expression referencing variable v and containing path p and *colf* denotes an iterator expression. (Let us note that we do not consider the OCL construct *let*.) In that case, $\bar{p}$ is *superior to* p in the hierarchy and we will denote $\bar{p}$ as *superior-path*(p). The translation of the paths in O proceeds from the paths at the higher levels of this hierarchy to the lower levels starting at the top-level.

Let us now discuss the translation of each single path p in O, s.t. p has steps $s_0, \ldots, s_m$. The pseudo-code of the translation algorithm is depicted in Alg. 3. The translation starts with step s_0, which remains unchanged. It establishes so called *translation context* for the next step s_1. The translation context is always a class in $\mathcal{S}'$. When $s_0 = self$, the translation context is the context class C' of O'. In other cases the translation context is the class targeted by the translation of *superior-path*(p). The targeted class is located during the translation of *superior-path*(p). The exact mechanism of how the targeted class is located during the translation is described in the following paragraphs.

The translation algorithm then proceeds consecutively through steps $s_1, \ldots, s_m$. Each step s_i is translated as follows. Let a class C'_i be the translation context of the step s_i. We locate the represented PIM class $C_i = I(C'_i)$. The step s_i specifies an attribute of C_i or an association endpoint of an association R_i connected to C_i. We will denote the attribute or association endpoint with X_i. When X_i is an attribute of C_i then we locate an attribute $X'_i \in \mathcal{S}'_a$ of C'_i which represents X_i. When X_i is an association endpoint of an association R_i then we locate an association $X'_i \in \mathcal{S}'_a$ which represents R_i and connects C'_i with another class C'_{i+1}.

When the located X'_i is an attribute we replace s_i with the name of X'_i and the translation of p is done. When X'_i is an association and *parent*(X'_i) = C'_i we replace s_i with the name of X'_i or the name of *child*(X'_i) (when X'_i does not have a name). When *child*(X'_i) = C'_i we replace s_i with the reserved word `parent`. When $i = m$ the translation is done and C'_i is *the class targeted by the translation of* p. When $i < m$ the translation proceeds to s_{i+1} and the other class C'_{i+1} becomes the translation context for the next step s_{i+1}.

Let us now demonstrate the direct translation algorithm on the PSM schema depicted in Fig. 5.2 and recall the OCL expression O over PIM schema depicted in Fig. 5.1 which we have already discussed in Section 5.4.1:

context Organization **inv** PIM_IC1:
self.Department.Team−>forAll(t |
  t.member−>size() < 0.1 ∗ self.Department.employee−>size())

First, the algorithm identifies the context class of O'. It is the PSM class `Organization'` which represents the PIM class `Organization` which is the context class of O. Then it translates all navigation paths in O, i.e.,

- p_1: «*self.Department.Team*»
- p_2: «*t.member*»
- p_3: «*self.Department.employee*»

The paths p_1 and p_3 start with the context variable *self* and are therefore at the top level of the hierarchy of paths in O. The path p_2 starts with the variable `t` which is declared in the collection function *forAll* which follows p_1. Therefore *superior-path*(p_2) = p_1. The algorithm firstly translates paths at the top level of

the hierarchy, i.e., p_1 and p_3. For both, the initial translation context is the PSM class `Organization′`. The result is

- p'_1: «*self.dpt.tm*»
- p'_3: «*self.dpt.emp*»

Then, it translates p'_2. Its initial translation context is the class targeted during the translation of p'_1, i.e., `Team′`. The resulting path is as follows:

- p'_2: «*t.mem*»

The algorithm translates only the context class and the navigation paths in O. The rest is not translated. The resulting PSM OCL O' constraint is:

context Organization **inv** PSM_IC1:
self.dpt.tm−>forAll(t | t.mem−>size() < 0.1 ∗ self.dpt.emp−>size())

The direct translation also succeeds in translating the constraint PIM IC2 from Figure 5.1:

context Organization **inv** PSM_IC2:
self.budget <= self.dpt.budget−>sum()

5.4.2 Problems with Direct Translation and its Improvements

The results of the direct translation algorithm are correct only in certain cases. Let us now analyze situations when the direct translation does not work correctly.

The first problem is that a PIM class $C \in \mathcal{S}_c$ may have more different representations $C'_1, \ldots, C'_k \in \mathcal{S}'_c$. Therefore, when C is the context class of O it is hard to decide which of the representations should be the context class of O'. The ideal option is to choose a representation C' whose attributes and associations represent all attributes and associations of C restricted by O. However, such representation may not always exist.

Let us demonstrate the problem on the PSM schema depicted in Fig. 5.2 and the following OCL expression O over the PIM schema depicted in Fig. 5.1 (it specifies that an employee can not be on its internship in a department which is his or her employer):

context Employee **inv** PIM_IC3:
self.internship <> self.employer

The context class of O is `Employee`. O restricts two associations of `Employee`: {`Employee`, `employer`} and {`intern`, `internship`}. There are three different representations of class `Employee` in the PSM schema: `Member′`, `Employee′`, and `Intern′`. We have to decide which of them will be the context class of the resulting O'.

It is not meaningful to choose class `Member′` as the context class. Its associations do not represent anything from the two associations constrained by O. On the other hand, the parent association of `Employee′` represents association {`Employee`, `employer`}. The parent association of `Intern′` represents {`Employee`, `internship`}. At the same time however, the former is missing an association which would represent {`Employee`, `internship`} and the other is missing an association representing {`Employee`, `employer`}.

As the example shows, automatic selection of the context class of O' can not be achieved, because they may be several candidates, evenly suitable. As a first heuristic, we may restrict the candidates only to those which are meaningful. A candidate C' is *meaningful* only when its attributes and associations represent some of the attributes and associations of C restricted by O. Formally, there must exist a navigation path $p = s_0.\ldots.s_n$ in O s.t. $s_0 =$ `self` and one of the following conditions must be satisfied:

- If s_1 leads to an attribute A then there is an attribute $A' \in \mathit{attributes}(C')$ s.t. $I(A') = A$.
- If s_1 leads to an association end E_1 of an association $R = \{E_1, E_2\}$ then there is an association $R' \in \mathit{associations}(C')$ s.t. $I(R') = (E_1, E_2)$ or $I(R') = (E_2, E_1)$.

(Let us note that a step of a navigation path always leads to an attribute or association end because of the restrictions to the syntax of navigation paths given in Section 5.4.1). It is clear that there may exist several different meaningful candidates C'_{cand-1}, ..., C'_{cand-m}. However, we are not able to choose the candidate automatically (in the presented case, neither candidate is better). We therefore need a user who decides which candidate will be chosen. In our sample, we have two candidates: `Employee′` and `Intern′`. The user may choose any of them as the context of the resulting translation.

The chosen candidate C' may be insufficient because its attributes and associations may not represent all attributes and associations of C restricted by O. In other words, there may be a navigation path $p = s_0.s_1 \ldots.s_n$ in O s.t. it is not possible to translate the step s_1 with Alg. 3. This is because it is not possible to navigate from the chosen translation context C' according to s_1.

In that case, it is necessary to change the current translation context to another PSM class D' s.t. $I(D') = C$. The attributes and associations of D' must represent some of the attributes and associations of C restricted by O. A similar situation may occur later during the translation of a step s_i of p because it is not possible to navigate from the translation context C'_i of s_i as specified by s_i. Again, we need to change the translation context to class D'_i similarly.

In both scenarios, the translation context is changed by applying the collection function toD'_i. This means appending $.\,\mathit{toD}'_i()$ to the end of the already translated part of O before the translation of s_i itself.

Similarly to the problem of choosing the correct context class, it may be a problem to choose the correct D'_i. Again, when there are more possibilities, we need a user who makes the selection.

As a demonstration, suppose the PSM schema depicted in Fig. 5.3a. In this schema, interpretation $I($`Department′`$) = I($`Host′`$) =$ `Department`. We will attempt to translate the following OCL expression O over the PIM schema depicted in Fig. 5.1 (it specifies that project teams may be hosted only by departments in a sponsoring organization):

context Project **inv** PIM_IC4:
self.Team−>forAll(t | t.host.Organization = self.sponsor)

In this case, the direct translation does not work. It automatically selects class `Project′` as the context class of O'. (There is no other class in the PSM schema representing `Project`.) O contains following navigation paths:

– p_1: «*self.Team*»
– p_2: «*t.host.Organization*»
– p_3: «*self.sponzor*»

The path p_1 is translated to a path «*self.team*». Its translation targets the class `Team′`. Because p_1 is the superior path to p_2, the translation of p_2 starts with the translation context set to `Team′`. The translation of the first step of p_2 `host` moves from `Team′` to `Host′` . From here it needs to move to `Department′`. However, there is no association in the PSM schema connected to `Host′` representing the PIM association connecting the PIM classes `Department` and `Organization`. Therefore, we need to change the translation context to class `Department′` whose parent association represents the PIM association. For this we call the collection function *toDepartment*. The last step of p_2 is then translated to a step navigating from `Department′` to its parent class `Organization′`.

The path p_3 is translated directly and the resulting translation of O is

context Project **inv** PSM_IC4:
self.team−>forAll(t | t.host.toDepartment().parent = self.parent)

Changing the translation context is also necessary when O compares instances of the same PIM class which are targeted by two different navigation paths in O. Suppose the following OCL expression O over the PIM schema depicted in Fig. 5.1

context Department **inv** PIM_IC5:
self.Employee.Team.host−>forAll(h | h = self)

It specifies that teams of employees of a given department can be hosted only by that department. It contains two navigation paths:

– p_1: «*self.Employee.Team.host*»
– p_2: «*self*»

Both target class `Department` in the PIM schema but through two different navigation paths p_1 and p_2. Suppose the PSM schema depicted in Fig. 5.3b. In this schema interpretation $I(\texttt{Department}') = I(\texttt{Employer}') = I(\texttt{Internship}') =$ `Department`. The direct algorithm translates O to the following PSM OCL expression O':

context Employer
inv: self.parent.parent.host = self

During the translation, the user had to decide that the context class of O' will be `Employer′` because there were two possible context classes (the other is `Department′`). The resulting OCL expression is not correct because it compares instances of the target classes of two different navigation paths in the PSM schema: `Department′` and `Employer′`. Instances of two different classes are not comparable. Therefore, one of the navigation paths needs to be supplemented with *to* function. With this function, the resulting OCL expression compares instances of the same class:

context Employer
inv: self.parent.parent.host.toEmployer() = self

(Let us note that this OCL expression is still not correct because there may be more different instances of class `Employer′` which represent the same instance of

the PIM class `Department`. Therefore, it compares a collection of instances with a single instance which is not correct. We will discuss this in the rest of this section.)

The second problem is the hierarchical nature of $\mathcal{S}'$ which may lead to redundant occurrences of instances of classes of $\mathcal{S}$ in XML documents. Suppose a class C' in $\mathcal{S}'$ which represents a class C in $\mathcal{S}$. As we explained in Section 3, C' specifies how instances of C are represented in XML documents. In other words, an instance of C' models an occurrence of the instance of C in XML documents. Let R' be the parent association of C' which represents an association $R = (E_C, E_D) \in \mathcal{S}_r$, $participant(E_C) = C$, $participant(E_D) = D$ (association R connects classes C and D in $\mathcal{S}$). Let the maximal cardinality of E_D in R be $m..n$ st. $n > 1$. In that case, there may be more different instances of C' in the same XML document modeling an occurrence of the same instance of C. We call this situation that C' *leads to redundant occurrences of* C.

The problem with redundant occurrences is that a PIM navigation expression can not be directly translated to a corresponding PSM navigation expression where the direction of navigation is upwards. Association $\texttt{Team}' - \texttt{Employee}'$ in the schema in Fig. 5.3b is an example of this issue. When t is an instance of $\texttt{Team}'$, expression `t.member` returns all members of the team. However, when e is an instance of $\texttt{Employee}'$, expression `m.parent` does not return all teams having m as a member. It returns only one team (because in XML, a node has only one parent).

In the general case, C' leading to redundant occurrences of C complicates the translation of O, if O navigates via R from C to the other end D. For a given instance of C O retrieves a collection of all associated instances of D. On the other hand, O' navigates from an instance of C', i.e., from one occurrence of the instance of C, in the upwards direction via R'. Because of the hierarchical structure, an instance of C' has only one parent instance. Therefore, O' navigates only to an occurrence of one associated instance of D instead of occurrences of all associated instances of D and the directly translated O' from O does not work correctly.

For demonstration, suppose the PSM schema depicted in Fig. 5.3b and the following PIM OCL expression O over the PIM schema depicted in Fig. 5.1 (it specifies that an employee can not be a member of more than 5 teams):

context Employee **inv** PIM_IC6:
self.Team−>size() < 5

This simple OCL expression is translated by the direct algorithm to the following PSM OCL expression O':

context Employee
inv: self.parent−>size() < 5

The problem is that while O works with an instance of `Employee`, O' works with an occurrence of that instance. Because of the cardinality of `Employee` in the association, there may be more different occurrences of the instance. Therefore, O' does not work correctly – it cannot count the number of teams an employee is a member of.

Another complication caused by redundancies may appear when O compares instances of the same PIM class which are targeted by two different navigation

paths in O. We have already discussed this kind of OCL expressions previously and we have used the following sample expression:

context Department **inv** PIM_IC5:
self.Employee.Team.host−>forAll(h | h = self)

We have shown that the direct algorithm extended with function *to* results to the following OCL expression O' (the user chooses `Employer′` as the context class):

context Employer
inv: self.parent.parent.host.toEmployer() = self

The problem is that `Employer′` leads to redundant occurrences of `Department`. Therefore, function *toEmployer* leads to a collection of more different instances of `Employer′` which represent the same instance of `Department`. However, O' compares this collection with a single instance of `Employer′` assigned to `self` variable.

On the base of the previous discussion we can see that the navigation principles of PIM OCL expressions are different from those of PSM OCL expression. This might cause that some directly translated PSM OCL expressions are not correct.

A direct simple solution to this problem is to forbid redundancies. More precisely, a navigation path can not navigate in a PSM schema in the upwards direction via an association where the maximum cardinality of the child in the association is greater then 1. In that case, the direct translation is not performed and a user is notified.

A more advanced but more complex solution is to extend the direct translation algorithm so that it translates not only the context class and navigation paths in PIM OCL expressions to their PSM equivalents but also works with other OCL constructs. However, this is out of scope of this paper and we leave it as our future work. Let us only demonstrate this extension on a few examples.

First, let us again consider the last PIM OCL expression and its translation. The solution of the problem of comparing the collection of instances of `Employer′` with a single instance is to change the comparison operator `=` to a collection function `forall`. The resulting PSM OCL follows.

context Employer
inv: self.parent.parent.host.toEmployer()−>forall(h | h = self)

Second, let us again consider the PIM OCL expression which restricts the number of teams an employee can participate in:

context Employee **inv** PIM_IC6:
self.Team−>size() < 5

The direct translation algorithm could not translate this expression to a PSM OCL expression over the PSM schema depicted in Fig. 5.3b as we have already discussed. The solution is to completely change the logic of the expression at the PSM level so that it moves only in the top-down direction. Therefore, the problems with redundancies do not occur. We need to change the context to a class whose instance contains all instances of `Team′`. This class is `Teams′`. From here, we need to get to all contained team members. For each member, we need to check whether the number of teams which contain the member is lower than 5. Because we are at the PSM level, each member can be represented by more different instances of `Employee′`. Therefore, we will not compare instances of `Employee′`, but we utilize the key for `Employee` (`empNo`) instead:

```
context Teams inv PSM_IC5:
self.team.member->forAll(m |
    self.Team->select(t | t.member.empNo->includes(m.empNo))->size() < 5)
```

Let us finally discuss the time complexity of the translation algorithm. The time complexity of the direct translation algorithms is linear with respect to the total length of navigation paths in a given PIM OCL expression. However, as we have already discuss, the direct translation does not work in many practical cases because classes in the PIM schema may have more different representations in the PSM schema. Then we have to apply rules which choose the correct representation for a given PIM class. Because of these rules the time complexity becomes $O(M * N)$ in the worst case where M and N are the number of classes in the PIM schema and PSM schema, respectively (for each PIM class C , we have to check all PSM classes which represent C). However, the worst case will be hardly achieved in practice. Usually, a PIM class has only one or few (i.e., $<< N$) representations in the PSM schema. Moreover, the amount of construct is not so high in real world schemas and, therefore, the time complexity of the translation is not restricting.

5.5 From PSM Level Expressions to Operational Level Expressions (PSM OCL → XPath)

In this section, we present an algorithm for translating PSM OCL expression O' into an XPath expression $X_{O'}$. We will now look at different kinds of OCL expressions in the OCL metamodel depicted in Figure 5.4. We will elaborate how they can be expressed equivalently in XPath. As we will show later in this section, for some classes of OCL expressions, a corresponding constructs/ functions in XPath do not exist. XPath by itself does not allow the user to define his own functions (apart from anonymous functions from version 3.0). However, other functions, besides those defined by XPath standard, can be added by the *host language*, which executes these expressions. We chose to XSLT as the host language and we created a library of XSLT functions, called *OclX*. Functions from *OclX* can be used in XPath expressions to provide sufficient expressive power. We chose XSLT, because it is the language used throughout our framework – XSLT transformations are created to adapt document (see Chpt. 4, 6) and XSLT can be used to validate Schematron schemas (more on this application in Sec 5.6.1). As an alternative, *OclX* might equivalently be defined using XQuery.

From now on, we will apply some restrictions on the class of considered OCL expressions. We will omit *StateExp* and *MessageExp*, since the notions of state and message(signal) have no counterparts in our domain (XML data). Due to space restrictions, we will also omit *TypeExps*, which deal with casting, and we will also treat all collections as sequences. Due to the architecture of XPath data model, we will also not allow nested sequences in expressions. We will get back to the problem of nested sequences and different types of collections in Sec 5.5.6. These conditions leave us with *LiteralExp*, *IfExp*, *VariableExp*, *LetExp*, two kinds of *LoopExps* (*IteratorExp* and *IterateExp*) and *FeatureCallExp* (which encompasses operations (and operators) and references to attributes and associations defined in the UML model). We also have to define consistent handling of variables.

Because our framework works with XML schemas tightly, we will use schema-aware expressions at the operational level. To execute them correctly, a schema-aware XPath engine is required, e.g., [76]. It is equally possible to translate into non-schema-aware expressions (and our implementation allows that as an option), but the resulting expressions are less readable, because they include a lot of casts and conversions of atomic types. That is why we use schema-aware expressions in the examples in this thesis.

5.5.1 Variables, literals, *let* and if expressions

Variables There are three ways a variable can be defined in an OCL expression. Each invariant has a context variable, which holds the validated object. It can be named explicitly (such as t in Figure 5.11) or, when no name is given, the name of the context variable is *self*. Iterator expressions (described later in this section) declare iteration variables (such as t in the expression «$self.team\text{->}forAll(t \mid \ldots)$)» in PSM IC4 in previous section). Let expressions (described later in this section as well) define a local variable.

We will construct the expression in such a way that the following principle holds:

Principle 1. *Every OCL variable used in O' corresponds to an XPath variable of the same name in $X_{O'}$. References to OCL variables (*VariableExp*) are translated as references to XPath variables.*

The OCL context variable (with default name *self* or named explicitly) is common in all invariants declared for the context. Therefore, to declare corresponding variable, we can utilize Schematron `sch:let` instruction in each rule (line `<sch:let name="t" value="."/>` in the example). Declaration of XPath variables for the other OCL variables (declared as a part of *LetExp* or *LoopExp*) will be created in accordance with Principle 1, as we will demonstrate later in this section.

LetExp defines a variable and initializes it with a value. The variable can be referenced via *VariableExp* in the subexpression of the given *LetExp*. XPath 3.0 added a corresponding construct – `let/return` expression. Thus, the following principle is in accord with Principle 1.

Principle 2. *Let O' be a* LetExp *expression «let $x : Type = initExp'$ in $subExp'$», where $initExp'$ and $subExp'$ are both OCL expressions and the latter is allowed to reference variable x. Than O' is translated to an XPath expression $X_{O'}$:* `let $x :=` $X_{initExp'}$ `return` $X_{subExp'}$*., where $X_{initExp'}$ and $X_{subExp'}$ are translations of $initExp'$ and $subExp'$ respectively.*

LiteralExp OCL allows literals for the predefined types, collection literals (e.g., «$Sequence\{1, 2, 3\}$»), tuple literals and special literals «*null*» (representing missing value) and «*invalid*» (representing erroneous expression).

Principle 3. *OCL literals are translated according to the following table.*

OCL	XPath
predefined type literal (literals for Real, e.g., «1.23», String, e.g., «'hello'» etc.)	corresponding XSD primitive type literal (`1.23`, `'hello'` etc.)
sequence literal, e.g., «$Sequence(1,2,3)$»	XPath sequence literal, e.g., `(1,2,3)`
tuple literal, e.g., «$Tuple\ \{\ name = 'John', age = 10\ \}$»	XPath map literal (more about tuples in Section 5.5.4)
«*null*» literal	empty sequence literal `()`
«*invalid*» literal	call of OclX function `invalid()` (more about error handling in Section 5.5.5)

IfExp Conditional expression in OCL has the same semantics as in XPath, it can be translated directly.

Principle 4. *Let O' be an* IfExp *expression «if cond' then thenExp' else elseExp'». Then O' is translated to an XPath expression $X_{O'}$:* `if (`$X_{cond'}$`) then` $X_{thenExp'}$ `else` $X_{elseExp'}$.

5.5.2 Translating Feature Calls

In the following subsections, we will show how we translate *FeatureCallExp*. There are two types of features in UML – *properties* and *operations*, which can be referenced via respective *FeatureCallExps*, as depicted in a separate diagram in Figure 5.4b. We will elaborate on both types – *PropertyCallExp* and *OperationCallExp* separately.

PropertyCallExp Examples of navigation expressions via *PropertyCallExp* are e.g., «$self.budget$» (in PSM IC2), «$e.internship$» (in PSM IC3) or «$self.team.member$» (in PSM IC6). The first one navigates to an attribute `budget'` of class `Organization'`, the second one navigates an association end `internship'` which is a part of the association between classes `Employee'` and `Department'`. Every *FeatureCallExp* has a *source* (inherited from *CallExp*, see Figure 5.4). The source in the first example is a *VariableExp* «$self$», in the second example, the source is a *VariableExp* «e». The third example is a chain of two *PropertyCallExps*. The source in the third example is a *PropertyCallExps* «$self.team$» whose source is a *VariableExp* «$self$». The whole navigation starts in class `Teams'`, goes through `Team'` and ends in `Member'`.

Navigation to properties can be translated by appending an XPath step which uses either the child or attribute axis. Translation of navigation to association ends depends on the direction of the association and whether the association has a name (an association without a name means that the subtree under the association is not enclosed by a wrapping tag, thus no XPath navigation is added). The following principle thus follows the rules from Table 3.1.

Principle 5. PropertyCallExp *is translated by appending an XPath step to the translation of the source expression. Let O' be a* PropertyCallExp *expression «source'.p'» and $X_{source'}$ be a translation of subexpression source'. If p' navigates to an attribute $A' \in \mathcal{S}'_a$ and $n' = name(A')$, then O' is translated to $X_{O'}$ as follows:*

$$X_{O'} = \begin{cases} X_{O'}\texttt{/child::}n' & \text{if } xform(A') = e \\ X_{O'}\texttt{/attribute::}n' & \text{if } xform(A') = a \end{cases}$$

If $R' = (E'_1, E'_2)$ *is an association and* p' *navigates to one of its ends* $E' \in \{E'_1, E'2\}$, O' *is translated as follows:*

$$X_{O'} = \begin{cases} X_{O'} & \text{if } name(R') = \lambda \\ X_{O'}\texttt{/child::}n' & \\ & \text{if } E' = E'_2 \wedge name(R') = n' \\ X_{O'}\texttt{/parent::node()} & \text{if } E' = E'_1 \end{cases}$$

OperationCallExp Application of predefined infix and prefix operators, calls of OCL standard library operations and calls of methods defined by the designer in the UML model all come under *OperationCallExp*. For a majority of predefined operators (such as «+», «and», etc.), a corresponding XPath operator exists as well. For those where no corresponding operator exists (e.g., «xor»), we provide a corresponding function in OclX library. (We do not include the exhaustive list in the paper. It can be found in the documentation for OclX [3].)

Principle 6. *Every* OperationCallExp O' *is translated into a call of corresponding operation/operator (with the same number of parameters; the translation of the source expression in* O' *becomes the first argument in XPath in* $X_{O'}$*, followed by the translation of the operation arguments in* O'*). The corresponding operation/operator is either a built-in XPath expression or it is defined in the OclX library.*

In Sec. 5.2, we introduced a new function $D'.toE'$, which is defined for each pair of PSM classes C'_1 and C'_2 with the same interpretation ($C = I(D') = I(E')$). This function call can be translated when there is a *key* defined for C and the key is represented both in D' and E'. The key can be utilized to locate the instances of C'_2 in the schema, as is described by Principle 7.

Principle 7. *Let* D' *and* E' *be a pair of PSM classes s.t.* $C = I(D') = I(E')$ *and let* $K_C = \{A_1, \ldots, A_n\} \subseteq attributes(C)$ *be a key for* C, $K'_D = \{D'_1, \ldots, D'_n\} \subseteq attributes(D')$ *a set of PSM attributes representing attributes from* K *and* $K'_E = \{E'_1, \ldots, E'_n\} \subseteq attributes(E')$ *s.t.* $I(D'_1) = I(E'_1), \ldots, I(D'_n) = I(E'_n)$.
OperationCallExp $O'_k = S'.toE'()$ *(where* S' *is of type «Collection(*D'*)») can be translated as* $X_{O'_k} =$ `(for $p in` $X_{S'}$ `return` $X_{A'}$`[$p/`$name(D'_1)$ `eq ./`$name(E'_1)$ `and ... and $p/`$name(D'_1)$ `eq ./`$name(E'_1)$`])`*, where* X'_A *is an XPath expression returning all instances of* E' *in the schema*[4].

To illustrate translation described in Prin. 7, we will show a translation of the constraint IC4, which we have already described earlier (project teams may be hosted only by departments in a sponsoring organization), for schema Fig. 5.3a. Constraint:

context Project **inv** PSM_IC4:
self.team−>forAll(t | t.host.toDepartment().parent = self.parent)

[3] OclX documentation: http://github.com/j-maly/oclx

[4] The syntax also presumes that all the key PSM attributes are represented as PSM elements (*xform* = *e*. For those attributes, which are not, attribute axes will is used to access them)

is translated as follows:

```
oclX:forAll($self/team, function($t)
    { (for $p in $t/host return //department[./name eq $p/name])/.. is $self/.. } )
```

The translation uses the key defined for `Department` which contains the attribute `Department.name`. The key is used to find the different representation of the same department.

5.5.3 Translating Iterator Expressions

Loop expressions, e.g., «$self.Team\text{->}forAll(t \mid t.host.Organization = self.sponsor)$» are archetypal for OCL – they perform the task of joins, quantification, maps and iterations. They are called using "->" operator. Instead of a list of parameters, the caller specifies the list of local variables and the *body* subexpression (see Figure 5.4). There are several important facts regarding loop expressions:

1. There are two kinds of *LoopExp*, a general *IterateExp* and *IteratorExp*. The general syntax of *IterateExp* is: «$iterate(i : Type; acc : Type = accInit \mid body\ )$», where i is the iteration variable, *acc* accumulator variable, *accInit* the accumulator initialization expression and *body* is an expression, which can refer to variables i and *acc*. The result is obtained by calling body expression repeatedly for each member of the collection (which is assigned to i and *acc* is assigned the result of the previous iteration). The value of *acc* after the last call is the result of the operation. The general syntax of *IteratorExp* is: «$iteratorName(i : Type \mid body\ )$», where *iteratorName* is one of the predefined OCL iterator expressions (such as *exists*, *closure*, etc.) or may be defined in a user extension, i is the iteration variable and *body* is an expression, which can refer to the iteration variable i (and all other variables valid in the place where the iterator expression is used). The semantics of the *IteratorExp* depends on the concrete iterator. The semantics for the predefined operators is given by the specification.

2. Except *closure*, all other predefined iterator expressions (and a majority of collection operations) can be defined in terms of the fundamental iterator expression *iterate*, e.g., «$exists(it \mid body)$» is defined as «$iterate(it;\ acc : Boolean = false \mid acc\ or\ body)$». Semantics of user-defined iterator expressions can be defined using *iterate* as well.

3. Iterator expressions *forAll* and *exists* (serving as quantifiers) together with boolean operators *not* and *implies* make OCL expressions at least as powerful as the first order logic. Operation *closure* increases the expressive power with the possibility to compute transitive closures. Operation *iterate* allows to compute primitively recursive functions (for more on the expressive power, see [48]).

4. Iterator expression *collect* is often used implicitly, because *PropertyCallExp* «source.property», where source is a collection (e.g., «team.member») is in fact a syntactic shortcut for «$source\text{->}collect(t \mid t.property)$» (e.g., «$team\text{->}collect(t \mid t.member)$»).

5. Multiple iteration variables, such as in «$c\text{->}forAll(v1, v2 \mid v1 <> v2)$», are allowed for some expressions, but that is just a syntactic shortcut for

nested calls:
«c->$forAll(v1 \mid c\text{->}forall(v2 \mid v1 <> v2)$)».

6. Collection operations define variables (iterators and accumulator) which are *local* (they are valid in the subexpression only). Other variables can be referenced from *body* expression as well – either the context variable (*self*) or variables defined by outer *LetExp* or *LoopExp* expressions. Variables except iteration variables (and accumulator in *iterate*) are *free* in the *body* expression.

For translation to XPath, property 2 implies that it is sufficient to show, how to translate *closure* and *iterate*, other operations can be defined using *iterate*. If we succeed, property 3 ensures that we can check constraints with non-trivial expressive power, incl. transitive closures. Property 5 relieves us from the necessity to deal with expressions with multiple iterators. However, property 6 implies that we have to deal with local variables for iterator expressions.

There is no operation similar to *iterate* in XPath. However, we will show that `iterate`, and consequently all the other iterator expressions, can be implemented as XSLT higher-order functions.

Higher-order functions (HOFs) are a new addition proposed for the drafts of the common XPath/XQuery/XSLT 3.0 data model, which introduces a new kind of item – *function item*. With function items, it is possible to:

1. assign functions to variables, pass them as parameters and return them from functions,
2. function items can be called,
3. declare anonymous functions in expressions.

HOF is a function, which expects a function item as a parameter or returns a function item as a result. OCL loop expression can be looked upon as HOF as well – they all expect a subexpression (*body*, see Figure 5.4), which is evaluated (called) repeatedly for each member of a collection. Property 6 mentioned above is important for the semantics – *body* subexpression can have free variables, which are, when evaluated, bound to variables defined in the *source* of the loop expression. E.g., in the expression PSM IC3 «$self.Team\text{->}forAll(t \mid t.host.Organization = self.sponsor)$», the *body* expression refers to two variables – $self$ and t. Variable t is the iteration variable, variable $self$ is free in *body*.

Principle 8. IterateExp *defines two variables, an* accumulator *and an* iteration variable. IteratorExp *defines one variable – an* iteration variable. *Translation of both* IterateExp *and* IteratorExp *must correspond to Principle 1, i.e., these variables must be available as XPath variables in the translation of the body expression.*

Figure 5.5 shows how *iterate* is implemented in OclX. It is a higher-order function, expecting a function item of two arguments in its third parameter `body`. The draft of XSLT 3.0 [93] introduces new instruction `xsl:iterate`, which we can use to our advantage. The function item is called repetitively for each member of the collection (line 10), with the two expected arguments – a member of the collection and the current value of the accumulator. When *body* was defined as an anonymous function item, the free variables it contains are bound to the variables

```
<xsl:function name="oclX:iterate" as="item()*">
  <xsl:param name="collection" as="item()*"/>
  <xsl:param name="accInit" as="item()*"/>
  <xsl:param name="body" as=function(item()*, item()*) as item()*/>

  <xsl:iterate select="1 to count($collection)">
    <xsl:param name="acc" select="$accInit" as="item()*" />
    <xsl:next-iteration>
      <xsl:with-param name="acc" select="$body($collection[current()], $acc)" />
    </xsl:next-iteration>
    <xsl:on-completion>
      <xsl:sequence select="$acc" />
    </xsl:on-completion>
  </xsl:iterate>
</xsl:function>

<xsl:function name="oclX:exists" as="xs:boolean">
  <xsl:param name="collection" as="item()*"/>
  <xsl:param name="body" as="function(item()) as xs:boolean"/>

  <xsl :sequence select="oclX:iterate($collection, false(),
      function($it, $acc) { $acc or ($body($it)) })" />
</xsl:function>
```

Figure 5.5: Functions `iterate` and `exists`

available in the calling expression, which is in accord with the semantics of loop expressions of OCL. The second part of Figure 5.5 shows how HOF *exists* can be defined in terms of HOF *iterate*. The definition utilizes an anonymous function node (line 23), which calls the function item passed as argument.

Principle 9. *Every* IterateExp *(call of* iterate*) is translated as a call of OclX HOF* `iterate`*. Every* IteratorExp *(call of some iterator expression, such as* exists *etc.) is translated as a call of an OclX HOF of the same name. OclX contains a HOF definition for each predefined iterator expression. Subexpression body is translated separately and the resulting* X_{body} *is passed as an anonymous function item to the HOF call.*

Some iterator expression can be in some cases translated using native XPath constructs without the need to call a HOF (as defined in Prin. 9), e.g., OCL *exists* function can be translated using `some/satisfies` expression. We discuss this sort of rewriting of queries in Sec. 5.5.8.

To conclude the part about the iterator expressions, we will address the operation *closure*. The syntax for *closure* is the same as for other iterator expression, but the difference is that the semantics of *closure* is not defined in terms of *iterate* – whereas the amount of iterations needed to compute *iterate* is fixed, *closure* computes a transitive closure of the *body* subexpression (the resulting collection must be in depth first preorder) – thus, it is not known, how many calls of *body* will be required. Again, there is no equivalent construct in standard XPath. The implementation of *closure* in OclX is depicted in Figure 5.6.

```
<xsl:function name="oclX:closure" as="item()*">
 <xsl:param name="collection" as="item()*"/>
 <xsl:param name="body" as="function(item()) as item()*"/>
 <xsl:sequence select="oclXin:closure-rec(reverse(collection), (),body)"/>
</xsl:function>

<xsl:function name="oclXin:closure-re" as="item()*">
 <xsl:param name="toDoStack" as="item()*"/>
 <xsl:param name="result" as="item()*"/>
 <xsl:param name="body" as="function(item()) as item()*"/>

 <xsl:choose>
  <xsl:when test="count($toDoStack) eq 0">
   <xsl:sequence select="$result"/>
  </xsl:when>
  <xsl:otherwise>
   <xsl:variable name="i" select="$toDoStack[last()]" as="item()"/>
   <xsl:variable name="contribution" select="$body($i)" as="item()*"/>
   <xsl:sequence select="oclXin:closure-rec(
     ($toDoStack[position() lt last()], reverse($contribution)), ($result, $i), $body)" />
  </xsl:otherwise>
 </xsl:choose>
</xsl:function>
```

Figure 5.6: Function `closure`

5.5.4 Tuples

In this subsection, we show how OCL expressions using tuples (anonymous types) can be translated to XPath. OCL allows the modeller to combine values in expressions into tuples. Tuples have a finite number of named parts and are created using *TupleLiteralExp*, a specialization of *LiteralExp*. An example of a tuple literal may be «*Tuple* { *firstName* = *'Jakub'*, *lastName* = *'Maly'*, *age* = 26 }». The values of the parts may be of arbitrary type, including collections and other tuples. The names of tuple parts (*firstName*, *lastName*, *age* in the example) must be unique and are used to access the parts of the tuple in expressions, similarly to attributes of classes (using "." notation), i.e., it is possible to write, e.g., «*employees*->*collect*(*e* | *Tuple* { *name* = *e.name*, *salary* = *e.salary* })->*select*(*t* | *t.salary* > 2000)» The result of this expression would be a collection of tuples. Tuples are also closely related to operation *product*, which computes a cartesian product of two collections:

```
product(c1:Collection(T1), c2:Collection(T2)) =
  self->iterate(e1; acc = Set{} | c2->iterate(e2; acc2 = acc |
    acc2->including(Tuple{first = e1, second = e2})))
```

The result of *product* is a collection of type «$Collection(Tuple(first : T1, second : T2))$», which contains all possible pairs where the first compound comes from collection $c1$ and the second from collection $c2$. This operation thus finalizes the suite of equivalents of the constructs required for a language to be relationally complete (see [14]):

1. Select - can be expressed using *select* iterator expression,
2. Project - can be expressed using *collect* iterator expression that creates a

tuple with the projected attributes (similarly as in the `employees` example above, which, in fact, performs projection to attributes name and salary) ,

3. Union - OCL has *union* operation as well,
4. Set difference - OCL has operation '-' working on sets,
5. Cartesian product - can be expressed using *product*,
6. (Rename) - can be expressed using *collect* in the same manner as project operation.

Thus, not only tuples can be used to write more concise expressions. Together with the operation *product*, they increase the expressive power of the language to relational completeness (see [48] for more on expressive power of OCL).

When we want to represent tuples in XPath data model, we have several options. We will discuss the disadvantages of the less suitable ones, because they also apply to representing collections, which we will discuss in Sec. 5.5.6.

One possibility is to use sequences, where each item in the sequence corresponds to one part of the tuple, i.e., the tuple from the example would be represented as sequence `('Jakub', 'Maly', 26)`. However, this solution is not completely satisfactory from the following reasons:

1. We lose "safety" and clarity in the expression, because we have to write `$t[1]` to represent the OCL expression «$t.firstName$».
2. When some part is missing (which is in OCL indicated by *null* value), we have to use some placeholder value to keep the length of the sequence constant (e.g., `('null', 'null', 26)`).
3. A part of a tuple in OCL can be of any type, including other tuples and collections. Here, this approach fails utterly, because all sequences are flattened in XPath. This also makes implementing product operation impossible, because it should return a collection of tuples, i.e., a collection of sequences.

Instead of representing tuples using sequences, an alternative would be to represent them using temporary documents, for example:

```
<Tuple>
    <firstName>Jakub</firstName>
    <lastName>Maly></lastName>
    <age>26</age>
</Tuple>
```

This approach would overcome the first issue («$t.firstName$» would be represented as `$t/firstName`), the second issue (an empty element could represent a missing part) and also the third issue (nesting is no problem here and collections could be encoded into trees as well). However, it brings two problems of its own:

1. The value which is about to become a part of a tuple, is copied to a temporary XML document. This would not hurt so much with atomic values, but would be a significant overhead, when the value was a node or a sequence of nodes in the input document (whole subtree would be copied in this case).
2. When a node is copied into a temporary document, the copy of the node is in no way connected to the original node. It would not be possible to

navigate outside its subtree (e.g., using parent axis, see Principle 5) and operations relying on node identity (such as the `is` operator) would give unexpected results when applied on nodes from the input tree and from the temporary tree.

XSLT 3.0 introduces another XPath item type – *map* item, which is a good fit for representing OCL tuples. A map item is an additional kind of an XPath item which was added in the Working Draft of XSLT 3.0 [93] (and is already implemented in [76]). Map items use atomic values for keys and allow items of any type as values. These properties of map items make them good candidates for representing tuples. Strings containing the name of a tuple part can be used as keys (and the names of parts must be distinct in an OCL tuple as well). The tuple from the example would be represented as `map{'firstName' := 'Jakub', 'lastName' := 'Maly', 'age' := 26}`, expression «$t.firstName$» would be represented as `$t('firstName')`. A value in a map can also be another map or sequence, which is consistent with semantics of OCL tuples. Operation product can be defined either by translating the definition from specification (using two nested *iterates*) or via a much more succinct XPath expression:

for e1 **in** *collection*1 **return for** e2 **in** *collection*2 **return**
map{'first' := e1, 'second' := e2}

Principle 10 summarizes translation of tuples.

Principle 10. *A tuple literal is translated into an XPath map item literal. Every tuple part is translated as a key/value pair in the item literal, the type of the key is string and the value of the key equals the name of the tuple part. Access to tuple parts is translated as indexing the tuple with a string corresponding to the accessed part.*

Neither of the examples in the previous section uses *LetExp* or tuples. We will demonstrate their usage on another example here. The expression PSM IC7 in Figure 5.1 verifies that an employee has at most two concurrent internships in different departments. The expression first computes an auxiliary variable internship, which contains a tuple for each employee in the organization. The tuple has two parts – *employee* and *departments* (the set of departments where the employee works as an intern). The type of *internships* is thus «*Set(Tuple(employee:Employee', departments:Set(Department')))*», which we abbreviate to *InternshipsSet* in the expression. The full translation of constraint PSM IC7 is depicted in Appendix C.1, together with the translations of the other constraints from the previous section.

```
context o:Organization inv PSM_IC7:
/* Only two internships allowed concurrently for one employee */
let internships : InternshipsSet = o.dpt.emp->iterate(e; acc: InternshipsSet = { } |
        acc->including(Tuple (
            employee = e,
            departments = o.dpt->select(d | d.int.empNo->includes(e.empNo)))))
  in /* now we work with the variable internships */
  internships->forAll(i | i.departments->size() < 3)
```

Listing 5.1: *LetExp* and *tuples* example

```
<xsl:function name="oclX:oclIsInvalid" as="xs:boolean">
 <xsl:param name="func" as="function() as item()*" />

 <!-- evaluate func and forget the result, return false -->
 <xsl:try select="let $result := $func() return false()">
  <xsl:catch>
   <xsl:if test="$debug">
    <xsl:message select="'Runtime error making the result invalid. '"/>
    <xsl:message select="' - code: ' || $err:code"/>
    <xsl:message select="' - description: ' || $err:description"/>
    <xsl:message select="' - value: ' || $err:value"/>
   </xsl:if>
   <!-- if function call fails, return true -->
   <xsl:sequence select="true()" />
  </xsl:catch>
 </xsl:try>
</xsl:function>
```

Figure 5.7: Implementation of *oclIsInvalid* using `xsl:try/xsl:catch`

5.5.5 Error Recovery

OCL as a language has a direct approach to "run-time" errors or exceptions. Errors in computation cause the result of the expression to be *invalid* – a special value, sole instance of type *OclInvalid*. It conforms to all other types (i.e., it can be assigned to any variable and can be a result of any expression) and any further computation with *invalid* results in *invalid* – except for operation *oclIsInvalid*[5]. It returns *true* when the computations results in *invalid* and *false* otherwise. This operation thus provides a very coarse-grained error checking mechanism available in OCL. Unlike OCL computation, XPath/XSLT 2.0 processor halts when it encounters a dynamic error and there is no equivalent of *oclIsInvalid*. It is also not possible to instruct it to jump to the validation of the next IC when a computation of one expression fails.

XSLT 3.0, however, introduces new instructions – `xsl:try` and `xsl:catch` – which provide means of recovery from dynamic errors. With these instructions, it is possible to implement *oclIsInvalid* as depicted in Fig. 5.7. We, again, utilize higher-order functions capabilities – the expression is evaluated in a function call wrapped in try/catch. OCL expression «*oclIsInvalid*(1/0)» can be translated to `oclX:oclIsInvalid(function() { 1 div 0 })`. Optionally, our validation pipeline (fully introduced in section 7) allows to safe-guard the evaluation of each expression using try/catch, so that the validation of another constraint may continue if a runtime error occurs and it is not contained by *oclIsInvalid*. In the debug mode, detailed info is given using `xsl:message`.

Principle 11. *Calls of functions* oclIsInvalid *and* oclIsUndefined *are translated into calls of corresponding OclX HOFs, implemented using try/catch instructions. Usages of* invalid *literal are translated into calls of* `invalid()`.

[5]To be accurate, another operation – *oclIsUndefined* – behaves equally to *oclIsInvalid* when the argument is *invalid*, but it also returns *true*, when the result of the computation is *null*

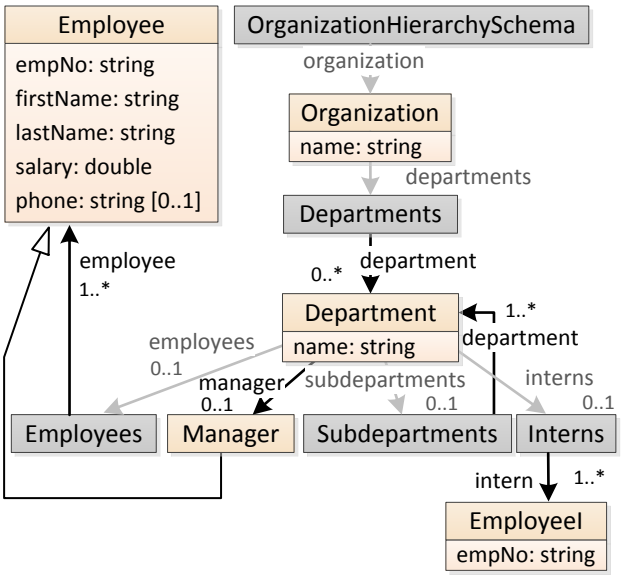


Figure 5.8: PSM schema – organization hierarchy

5.5.6 Collections

OCL defines an abstract type *Collection* and 4 different types of collections - *Set*, *OrderedSet*, *Bag* (in other languages sometimes called multiset) and *Sequence*. A member of any collection can be an arbitrary value, including another collection.

We treat all collections as sequences in *OclX*, yet it would be possible to represent the other kinds of collections using maps (or sequences as well).

Nested collections are a foreign concept to XPath data model. The disadvantages of encoding collections into temporary documents were discussed in Sec. 5.5.4.

With the introduction of maps, there is, a rather ugly, way of encoding nested sequences - thanks to the possibility of using maps as values and members of sequences. A nested sequence ((1,2),(3),()) could be encoded to:

map{'s' := (**map**{'s' := (1,2)}, **map**{'s' := (3)}, **map**{'s' := ()})}.

The expression returning number two would be written as:

(**map**{'s' := (**map**{'s' := (1,2)}, **map**{'s' := (3)}, **map**{'s' := ()})})('s')[1]('s')[2].

Using this approach also requires the functions which concatenate sequences not to use XPath operator ',', because it flattens the resulting sequence.

The double indexing (first to get the value from the map, second for getting the desired member of the sequence) can seem confusing, but it can be in fact hidden behind a library function at(i) (which OCL uses instead of the operator [i] to get an i-th member of a sequence). The wordy and a bit unclear way of creating a nested sequence appears in those expression that create nested sequences using literals. This, however, could be eliminated by using preprocessing of the schemas before evaluating the expressions.

5.5.7 Validation of Inheritance and Recursion

In this part, we examine the translation of OCL constraints regarding recursive structures and inheritance. Recursive structures are not unusual in XML data

```
context d:Department
/* PSM R1*/
inv: let count:Integer = d->closure(sd | sd.subdepartments.
  department).employees.employee->size()
in d.interns.intern->size() > 0 implies count >= 3
message: 'Only departments with at least 3 employees can accept interns,
department {d.name} has only less employees

context e: Employee
/* PSM E2 */
inv: e.empNo <> ''

context e: EmployeeI
/* PSM I1*/
inv: e.Interns.Department <> null
  implies e.Interns.Department <> e.toEmployee().Employees.Department
message: 'Internship in home department is forbidden'

context m:Manager
/* PSM M1 */
inv: m.Department.employees.employee.empNo->includes(m.empNo)
message: 'Manager is an employee of its department'
/* PSM M2 */
inv: m.phone <> null
message: 'Managers must state their phone numbers'
```

Figure 5.9: PSM ICs for organization schema

and are supported by all XML schema languages. Inheritance is less popular and supported only by XML Schema Language. On the other hand, inheritance is a core construct of UML/OCL, which also supports recursive structures in the form of associations whose association ends belong both to the same class.

Recursion and inheritance can be validated using OclX (by recursion we mean navigation along cycles in PSM schemas using closure). We will demonstrate them on a PSM schema in Fig. 5.8, the translation of this schema to an XSD can be found in Appendix A.8. The sample constraints are in Fig. 5.9.

Inheritance is a common feature in UML diagrams and OCL supports inheritance by allowing calls of inherited features (operations and properties, via *FeatureCallExp*) and rules of type conformance. The subexpression «*m.phone*» from Fig. 5.9 is legal because class `Manager'` inherits attribute `phone'` from class `Employee'`. The semantics of OCL also defines that invariants defined in the superclass apply also for all its subclasses. Thus, the invariant PSM E2 defined for class `Employee'` must also be met by instances of class `Manager'`.

At the PSM level, we support inheritance as well [35] – a class can inherit from another class which means that the element corresponding to the specific class will have all the attributes and subelements defined by the attributes and content of the general class (the inherited subelements come before the specific class' own subelements). This corresponds to the requirement that inherited features can be used in OCL feature call expressions.

When translating OCL expressions over a schema which contains a class hierarchy we must ensure that invariant O' defined for a superclass C are also checked for subclasses C_x. This can be achieved by:

1. copying the *rule* R obtained from translation of O' for every subclass. (It makes the resulting schema larger and less transparent.)
2. combining all the occurrences of C and C_x into the context of R using the XPath union operator (e.g., use the expression `//employee | //manager`). The translation of R is not repeated, but the resulting schema does not visibly show that the PSM schema utilizes inheritance.
3. utilizing the feature provided by Schematron for rule logic reuse – abstract patterns. Unlike the previous options, this one does not require the context variables to be named the same in the general and in the specific invariants.

We decided for the last option since it preserves the nature of inheritance. The rules for shared invariants are declared in abstract patterns and called by the patterns for derived classes. For every class participating in a generalization as a super class, an abstract pattern is generated. For every non-abstract class, which inherits from the class (and for the super classes themselves), an instance pattern is generated.

Principle 12. *Rules obtained by translation of invariants where the context is a class for which specialized classes exists are placed in abstract Schematron patterns. An instance pattern calling the abstract pattern is created for each subclass.*

Appendix C.4 shows a translation of invariants from Fig. 5.8. Constraint PSM E2 for class `Employee` is translated into an abstract pattern `Employee`. This abstract pattern is called both for instances of `Employee` and `Manager` (via patterns `Employee-as-Employee` and `Manager-as-Employee`).

The recursive association between departments (departments have subdepartments) is represented in the PSM schema in 5.8 by the cycle `Department-Subdepartments-Department`. This must be reflected in validation. The expression defining the context of rule PSM R1:

```
organization/departments/descendant::department
```

utilizes descendant axis. OCL constraints concerning recursive structures often utilize *closure* iterator expressions (see Sec. 5.5.3 earlier in this chapter), PSM R1 is an example of such constraint.

As a demonstration of translation, Appendix C contains translations of the expressions from the integrity constraints used in the examples in this section in the form of Schematron schemas. We describe how translation to Schematron works in 5.6.1 later in this chapter.

5.5.8 Operational Level Expression Rewriting

In Section 5.5.3, we showed that it is possible to translate iterator expressions to XPath using *OclX* functions. For every predefined OCL iterator expression, OclX defines a higher-order function which mirrors both the syntax and semantics of the iterator expression. Thus, the default translation (as defined by Principle 9) is syntactically closest to the original OCL expression. In some cases the usage of an iterator expression may be translated to a native XPath expression equivalent to the expression with higher-order functions. Such *rewritings* may be desirable for several reasons:

1. The resulting expression may be less complex, more readable and easier to understand for an XPath users.
2. The resulting expression may be less costly to evaluate, because XPath/XSLT processors are highly optimized for XPath axis evaluation. Also, every usage of anonymous functions has certain overhead.
3. By rewriting the expressions used in the schema, it may be possible to evade using OclX completely and the resulting schema would be a standard Schematron (XPath 2.0) schema, which may be used even by non-XSLT based Schematron validators.

In the following, we will expect $X_{collection'}$ and $X_{body'}$ ($X_{cond'}$, where more appropriate) to be translations of the expressions *collection'* (returning the source collection) and *body'*(or *cond'*, where more appropriate) (the body expressions of *IteratorExp*) respectively. Some rewritings may be used only for a certain class of expressions – these preconditions are given for each rewriting.

Definition 14 (x-safe)**.** *We will call an OCL expression* safe with respect to variable x *or* x-safe*, when it does not contain iterator expression referencing variable x.*

E.g., expression 1) «$x > z$->*select*$(y|y = 0)$» is *x-safe*, because x is not referenced in an iterator expression. Expression 2) «$z = y$->*select*$(u|x <> u)$» is not *x-safe*, because x is referenced in the iterator expression *select*. *X-safe* expression are an important subclass of expression when rewritings are concerned. When the body expression of an iterator expression is *safe with respect to the iterator variable*, references to the iterator variable can be replaced by references to context node in XPath filters. E.g., in the expression 1), the subexpression «$y = 0$» is *y-safe* and 1) can be thus translated as `$x > $z[. eq 0]`. When the body is not *y-safe*, it is not possible, because some iterator subexpression references y and in that occurence, y does not correspond to context node any more. We will denote $X_{exp'|x}$ the translation of *exp'* where all references to x are translated as references to context node `'.'`.

collect General form of collect is: «*collection'*->*collect*$(x \mid$ *body'*$)$» and it can be translated as follows:

1. `oclX:collect(`$X_{collection'}$`, function($x) { `$X_{body'}$` })`
2. `for $x in `$X_{collection'}$` return `$X_{body'}$
   This rewriting can be used in every case.
3. $X_{collection'}/X_{bodyR'}$
   Allowed when $X_{body'}$ is a PSM path starting in variable x, then $X_{bodyR'}$ is the path without variable x. This rewriting corresponds to OCL's syntactic shortcut for *collect*. Using this rewriting, it is possible to replace, e.g., the following expression: `oclX:collect($organization, function($o) { $o/department } )` with a more concise one `$organization/department`.

forAll/exists Expression exists/forAll returns true when at least one/every item in the source collection satisfies given condition. General form of forAll is: «*collection'*->$forAll(x \mid$ *cond'*$)$» and it can be translated as follows:

1. `oclX:forAll(`$X_{collection'}$`, function($x) { `$X_{cond'}$` })`
2. `every $x in `$X_{collection'}$` satisfies `$X_{cond'}$
   This rewriting can be used in every case. For *exists*, `some` will be used instead of `every`.

select/reject Expression *select/reject* returns the collection of those items from the source collection, which satisfy given condition. We will show rewritings for select, rewritings for reject can be obtained analogously after inverting the condition. General form of select is: «*collection'*->*select*(x | *cond'*)» and it can be translated as follows:

1. `oclX:select(`$X_{collection'}$`, function($x) { `$X_{cond'}$` })`
2. `for $x in `$X_{collection'}$` return if (`$X_{cond'}$`) then $x else ()`
   This rewriting can be used in every case.
3. $X_{collection'}$`[`$X_{cond'|x}$`]`
   Allowed when *cond'* is *x-safe*.
4. $X_{collection'}$`[let $x := .  return `$X_{cond'|x}$`]`
   This rewriting can be used as an alternative when *cond'* is not *x-safe* – variable `$x` is defined explicitly. However, `let/return` expression are only supported in XPath 3.0.

any Expression *any* returns one of the items from the source collection, which satisfy given condition (or *null* if no such item exists). General form of *any* is: «*collection'*->*any*(x | *cond'*)» and it can be translated as follows:

1. `oclX:any(`$X_{collection'}$`, function($x) { `$X_{cond'}$` })`
2. `(for $x in `$X_{collection'}$` return if (`$X_{cond'}$`) then $x else ())[1]`
   This rewriting can be used in every case. The result of evaluation is an empty sequence when no item satisfy the condition, which is consistent with representing *null* as na empty sequence (this also applies for the following rewriting).
3. `(`$X_{collection'}$`[`$X_{cond'|x}$`])[1]`
   Allowed when *cond'* is *x-safe*.
4. `(`$X_{collection'}$`[let $x := .  return `$X_{cond'|x}$`])[1]`
   Alternative when *cond'* is not *x-safe*, XPath 3.0 only.

one Expression *one* returns true if there is exactly one item in the collection satisfying given condition. General form of *one* is: «*col'*->*one*(x | *cond'*)» and it can be translated as follows:

1. `oclX:one(`$X_{col'}$`, function($x) { `$X_{cond'}$` })`
2. `count(for $x in `$X_{coll'}$` return if (`$X_{cond'}$`) then $x else ()) eq 1`
   This rewriting can be used in every case.
3. `count(`$X_{col'}$`[`$X_{cond'}$`]) eq 1`
   Allowed when *cond'* is *x-safe*.

4. `count(`$X_{col'}$`[let $x := . return` $X_{cond'}$`]) eq 1`
   Alternative when *cond'* is not *x-safe*, XPath 3.0 only.

closure The general form of *closure* is «*collection'*->*closure*(x | *body'*)». This general form can not be rewritten, but closure is often used to process hierarchical structures. When the hierarchical structure is also represented via element nesting in the XML document, XPath ancestor or descendant axes may be used. The rewritings are thus as follows:

1. `oclX:closure(`$X_{collection'}$`, function($x) { `$X_{body'}$` })`
2. $X_{collection'}$`/descendant-or-self::`$X_{bodyR'}$
   Allowed when *body'* is a PSM path and all navigation steps in the path are oriented *downwards*[6]. Expression *bodyR'* is a path containing only the last step in *body'*. As an example, the following expression, which is a translation of the constraint PSM R1 from Fig. 5.9:
   `oclX:closure(., function($sd) {$sd/subdepartments/department} )`
   can be rewritten into much more concise form:
   `./descendant-or-self::department`.
3. $X_{collection}$`/ancestor-or-self::`$X_{bodyR'}$
   Allowed when *body'* is a PSM path and all navigation steps in the path are oriented *upwards*[6]. Expression *bodyR'* is a path containing only the last step in *body'*.

iterate As we have stated in Section 5.5.3, standard XPath does not contain any expression corresponding to OCL *iterate* (general iteration with accumulator). XPath iterator `for ...in ...` has different semantics – it has no accumulator and one iteration has no access to the results of previous iterations. Thus, when *iterate* is used non-trivially, only the default translation using `oclX:iterate` extension is possible.

There are several other kinds of expressions our framework allows to rewrite, besides iterator expressions. E.g., a collection function «*at*(n)», which selects the n-th member of a sequence, can be translated either as a call of an *OclX* function: `oclX:at($collection,$n)`, or as an XPath predicate: `$collection[$n]`. Listings in Appendix C.5 shows translation of integrity constraints from Fig. 5.9 with rewritings applied. These translations are equivalent to the corresponding translations in Appendix C.4.

5.6 Applications of OCL for XML Data

To conclude this chapter, we will show how translated OCL constraints can be used in XML systems.

A straightforward application is to model queries at the conceptual level and translate them automatically to the operational level (using the algorithms from the previous two sections). The translated quires can be executed upon XML documents.

[6]The rewritten expression processes the *whole hierarchy*. When the original expression selects only a part of the hierarchy, the rewriting can not be used.

Another application of OCL in our framework is to verify invariants in XML documents. The user can define invariants at the PIM or PSM level and the system will automatically generate a schema to verify the constraints. If the document passes validation against the schema, the invariants are satisfied. We will use Schematron [29] for validation. Schematron schemas are built upon XPath expressions and we can thus apply the results of the previous sections, where we have shown how to translate from OCL to XPath. See more on this application in 5.6.1.

Using OCL, it is also possible to define new methods incl. implementation or define implementation of the methods already defined in the class diagram. Methods defined in OCL are 'pure' (they do not have any side-effects) and their body is an OCL expression. Such a method can then be used in every other OCL expression (in invariants etc.). We will show how OCL method definition can be translated into XSLT function definitions. See more on this application in 5.6.2.

In Chpt. 6, we will enhance the algorithm for document adaptation (Alg. 2).

5.6.1 Validation of Integrity Constraints Using Schematron

In this subsection, we describe how to translate a list of PSM OCL invariants attached to a single PSM schema into a Schematron schema, which can be used to validate the invariants in XML documents. A Schematron schema often complements[7] an XSD/RNG/DTD schema (which prescribes the overall structure of XML documents). Schematron is a straightforward rule-based language. It consists of rule declarations, where every portion of a document matching a *rule* (match patterns follow the same syntax as in XSLT templates) is tested for *assertions* defined in that rule. Assertions are expressed as XPath tests and an assertion is violated, when the *effective boolean value*[8] of the expression is *false*. The example rule in Fig. 5.10 tests, whether every element `person` has subelement `name`.

```
<rule context="//person">
    <assert test="name">Subelement 'name' is missing.</assert>
</rule>
```

Figure 5.10: Simple schematron rule

The usage of XSLT patterns for contexts of rules and XPath expression for tests of asserts was chosen because those are technologies well established in the XML ecosystem. It also facilitates Schematron validation using an XSLT processor – an XSLT pipeline can be used to translate a Schematron schema S into a validation XSLT transformation T_S[9]. T_S is executed upon a validated XML document and outputs a report about the progress and results of validation[10]

[7] As a matter of fact, the recommendation of XML Schema 1.1[92] allows to include some of the Schematron constructs directly in the XSD.

[8] The term *effective boolean value* is defined by XPath specification[89].

[9] http://schematron.com offers an implementation of XSLT pipeline to generate T_S for public use.

[10] Results produced by T_S are formatted using SVRL – Schematron Validation Report Language, which is part of Schematron specification [29].

OCL construct	**Schematron construct**
OCL script	Schematron schema
Constraint block	Pattern with a rule
Context classifier	Pattern id, value of context attribute
Context variable	`let` instruction for a variable
Invariant	Assert
Invariant body	**Expression in assert test**
Error message	Failed assert text
Subexpression in error message	`value-of` instruction in assert

Table 5.1: Translation of principal OCL constructs

comprising successfully checked constraints, violated constraints, and the locations of the errors.

The power of Schematron is thus determined by the power of XPath (or XPath running in XSLT context, when an XSLT-based validator is used), because XPath expressions are used in tests. Since *OclX* is implemented using pure XSLT, our approach does not require modification in Schematron validators – if the validator uses XSLT internally, it's logic can be preserved providing that T_S imports *OclX* library (details of the validation workflow are described later in Chpt 7).

Table 5.1 outlines the rules for translation of a list of invariants into Schematron constructs. Following these rules, we obtain a skeleton of the Schematron schema S for a PSM schema $\mathcal{S}'$.

It is apparent that *rules' contexts* and *asserts* in Schematron play the same role as *contexts* and *invariants* in OCL. Thus, by creating a rule for each OCL context declaration and adding an assert in the rule for each invariant, we can create a schema verifying the validity of the PSM invariant. The core of the algorithm – translating the invariant's body O' into an XPath expression $X_{O'}$ – was already described in Sec 5.5. What is left is to define the context variable in concord with Principle 1 – Schematron `let` instruction is used for the purpose. Fig. 5.11 shows a concrete example of the translation.

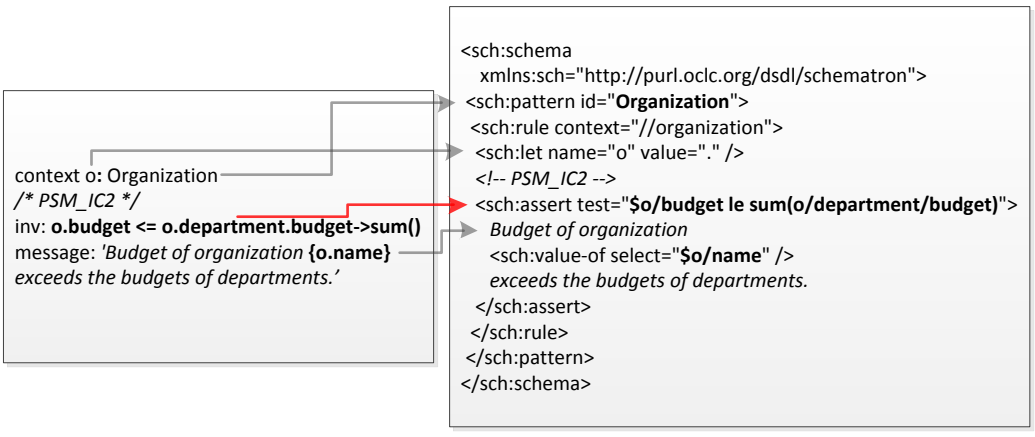


Figure 5.11: Example of translation of principal OCL constructs

5.6.2 Translating OCL Function Definitions

Methods can be defined solely in OCL and used from OCL expressions afterwords. Both static and instance(non-static) methods can be defined using operation body expressions. Examples of both are depicted in Fig. 5.12. The first one (PSM DO1) is a static method checking, whether two specified date intervals overlap. The second one (PSM DO2) adds a function that allows to find all departments where an employee occupies a post of an intern. This function is used in an invariant (PSM IN1) to check that no employee has more than two concurrent internships. Note that PSM IN1 utilizes *toEmployee* – a traversal function introduced in 5.2 to get `EmployeeI` classes from `Employee` class.

Since it is not possible to define named functions in XPath, we will use XSLT to define the new functions at the operational level (Similarly as we did with *OclX libarary*.

OCL operation definitions and calls are naturally translated to XSLT function definitions and calls. Depending whether the function is static or not, the method gets an additional parameter standing in for the context variable. The translation is formalized by Principle 13 and sample translations are depicted in Fig. 5.13.

Principle 13. *Each operation M' of class C' defined using OCL definition with expression O' is translated into an XSLT function declaration $F_{M'}$. The name of the function is name(C')-name(M'). If M' is not static, the first parameter of $F_{M'}$ corresponds to the context variable. The parameters of M' are translated as the remaining parameters of $F_{M'}$. Every call of M' is translated as a call of $F_{M'}$ and the instance of C' is passed as the first parameter to $F_{M'}$ when M' is not static. The remaining parameters in the function call are translated as subexpression. Function $F_{M'}$ returns the value of the expression $X_{O'}$ obtained by translating O'.*

```
context Date
/*PSM DO1*/
static def: isOverlap(d1from : Date, d1to : Date, d2from : Date, d2to : Date) : Boolean =
  if (d1from > d1to or d2from > d2to) then invalid /* check inputs */
  else (d1from < d2todB) and (d2from < d1to)

context e:Employee
/* PSM DO2 */
def: getInternshipDepartments() : Set(Department) =
  e.toEmployeeI().parent.Department

context e:Employee
/* PSM IN1 */
inv: e.getInternshipDepartments()->size() < 3
message: 'Only two internships allowed concurrently for one employee'
```

Figure 5.12: Method definitions in OCL

```
<!-- translations of OCL-defined operations,
       this goes into XSLT user functions file -->
<xsl:function name="user:Date-isOverlap" as="xs:boolean">
  <xsl:param name="d1from" as="xs:dateTime" />
  <xsl:param name="d1to" as="xs:dateTime" />
  <xsl:param name="d2from" as="xs:dateTime" />
  <xsl:param name="d2to" as="xs:dateTime" />
  <xsl:sequence select="if (d1from &gt; d1to or d2from &gt; d2to)
    then oclx:invalid() /* check inputs */
    else (d1from &lt; d2to) and (d2from &lt; d1to)" />
</xsl:function>

<xsl:function name="user:Employee-getInternshipDepartments" as="xs:item()*">
  <xsl:param name="e" as="item()" />
  <xsl:sequence select="let $p := $e return //intern[./id = $p/id]/../.." />
</xsl:function>

<!-- schematron schema with invariant using the ocl-defined operation -->
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron">
 <sch:pattern id="EmployeeO">
  <sch:rule context=
      "/organization/departments/descendant::department/employees/employee">
   <sch:let name="e" value="." />
     <sch:assert test="count(user:Employee-getInternshipDepartments($e)) &lt; 3">
       Only two internships allowed concurrently for one employee</sch:assert>
  </sch:rule>
 </sch:pattern>
</sch:schema>
```

Figure 5.13: Translation of the OCL-defined operatoins as XSLT functions

6. Document Adaptation with Semantic Annotations

In this chapter, we will extend our adaptation approach from Chpt. 4. We will focus on the semantics of more complex changes, where mere mapping between structures does not provide sufficient information. In this chapter, we will apply the results from the previous chapter – translation of OCL expressions into XPath expressions. In this chapter, we will show how to use the expressions to pull and transform the data from the adapted document. This is an enhancement of version links described in Chpt. 4 – sometimes, connecting the constructs in the old and new version with a version link does not fully describe the relationship between the two version. For this cases, we propose *annotations*, which have the form of formal OCL expressions over the old and new version of the model.

6.1 Requirements for Semantic Adaptation

In Chpt. 4, we introduced a finite set of change predicates, that can be tested for the old and new versions of the schema to detect change instances. The proposed set of predicates can be used to detect all changes in the structure of documents. However, when we want to generate the adaptation script, we need not only to adapt the structure, but also the content. Since content is not described in the PSM schema, detected change instances do not always provide us with enough information to adapt the documents accordingly. We will demonstrate this using the example in Fig. 6.1. It shows two versions of a PSM schema which models a history of customer's purchases in an e-commerce system, together with sample XML documents. The following changes were made by the designer to the schema in version v:

1. Attribute `name` is replaced by a pair of attributes `firstName` and `lastName`. The value of `name` in v consists of first name and last name separated by space character, in $\tilde{v}$, they are represented as separate elements.

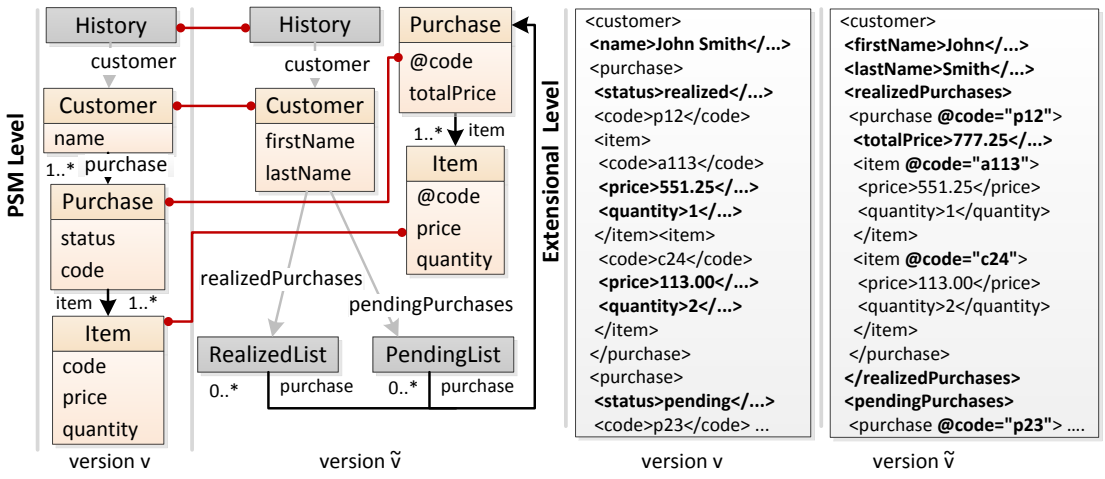


Figure 6.1: Two versions of customer history schema

```
<xsl:stylesheet version='3.0' xmlns:xsl=
  'http://www.w3.org/1999/XSL/Transform'>
 <xsl:template match='/customer'>
  <customer>
   <firstName/>
   <lastName/>
   <xsl:call-template name='RealizedList'/>
   <xsl:call-template name='PendingList'/>
  </customer>
 </xsl:template>

 <xsl:template name='RealizedList'>
  <realizedPurchases/>
 </xsl:template>
 <xsl:template name='PendingList'>
  <pendingPurchases/>
 </xsl:template>

 <xsl:template match='/customer/purchase'>
  <purchase>
   <xsl:apply-templates select='code' />
   <totalPrice/>
   <xsl:apply-templates select='item' />
  </purchase>
 </xsl:template>

 <xsl:template
   match='/customer/purchase/item'>
  <item>
   <xsl:apply-templates select='code'/>
   <xsl:apply-templates
     select='price'/>
   <xsl:apply-templates
     select='quantity'/>
  </item>
 </xsl:template>

 <xsl:template priority='0'
   match='item/code | purchase/code' >
  <xsl:attribute name='{name()}'>
   <xsl:value-of select='.' />
  </xsl:attribute>
 </xsl:template>

 <xsl:template priority='0'
   match='price | quantity' >
  <xsl:copy-of select='.' />
 </xsl:template>
</xsl:stylesheet>
```

Figure 6.2: Stylesheet for syntactic (structural) adaptation

2. A new attribute `totalPrice` is added to `Purchase` class. The value of `totalPrice` must be the sum of prices of all the purchased items.
3. The list of purchases is divided (based on the value of `status`) into two wrapping PSM classes – `realizedPurchases` and `pendingPurchases`.
4. All `codes` in the schema now model XML attributes instead of XML elements (*xform* is changed from e to a).

Algorithm 2 will be able to correctly adapt change (4) and will create the structure for changes (1)-(4)(elements `realizedPurchases`, `pendingPurchases`, `firstName`, `lastName` and `totalPrice`). The substantial part of the resulting adaptation script is depicted in Fig. 6.2. This script is capable of adapting to the structural changes, but it does not contain any information about how to:

1. Initialize the new attributes (`Customer.firstName`, `Customer.lastName`, `Purchase.price`).
2. Create the subtrees corresponding to added classes – associations between classes `RealizedList-Purchase` and `PendingList-Purchase` both have cardinality $0..*$ and since the adaptation algorithm always chooses the lower cardinality when creating new content, the elements will be created empty in the adapted document. The stylesheet in Figure 6.2 does contain a template for `Purchase`, but it is never applied.

The algorithm has no way of telling how to fill the created elements with values/content merely from the version links themselves, without any additional information.

In this chapter, we will show how to add semantic annotations to the version links and schemas. Semantic annotations allow the modeller to define the relationship between the *contents* of the documents in the old and new version of the

schema. We extend Alg. 2 by providing the user with a way to refine the default behaviour of Alg. 2 at the conceptual level, formally and without the need to use the implementation language.

6.2 Extending OCL to Define Relationships In Versioned Model

To formally express the relations between the content of the old and adapted version of the document, we will again use OCL. E.g the second change from the motivational example can be described using an equation:

totalPrice = item->collect(price * quantity)->sum()

and saying that the equation must be true for all instances of `Purchase` after the document is adapted. However, we phrased the condition using only *the terms of the adapted version* schema and the values in the adapted document. Our goal is to define the new value of the attribute primarily in *the terms of the original version* of the schema and the values in the original document. This is more evident for the first change in the example – to define the new value of the attributes `firstName` and `lastName`, we must refer to the value of `name` in the original document. Standard OCL does not provide any means to write expressions over multiple versions of the model. In this section, we will extend OCL to allow that.

In general, to define the value of attribute $\widetilde{A'}$ of class $\widetilde{C'}$, we will use an expression $init_{\widetilde{A'}}$ and an attribute initialization constraint, which is defined by the OCL specification:

context $\widetilde{C'}$::$\widetilde{A'}$
init: $init_{\widetilde{A'}}$

The notation above signifies that $init_{\widetilde{A'}}$ is evaluated and its result is assigned as a value of attribute $\widetilde{A'}$. Attribute initialization expression is one of the *annotations*, we support in our model (another one – association end initialization expression – will be introduced later in this section). We will use standard OCL syntax in the paper, but an implementation of this approach might choose another way to enter/display these annotations, e.g. in some kind of details view for an attribute, on mouse hover or directly in the diagram, similarly as comments in UML are usually displayed. OCL allows the expression $init_{\widetilde{A'}}$ to refer to the context variable of type $\widetilde{C'}$ and named *self* by default. From the context variable, the rest of the model can be navigated. This does not wholly fit our needs, because during adaptation, we need to refer to and pull the necessary information from the *previous version* of the model. For this purpose, we add a new operation *prev*.

Definition 15. *Function* prev() *can be called on any instance (object) of class* $\widetilde{C'}$ *in the target schema, which is linked to class* C' *in the source schema via a version link* $(C', \widetilde{C'}) \in \mathcal{VL}$. *The return type of* $\widetilde{C'}$.prev() *is then* C'. *Calling* prev() *returns the instance of* $\widetilde{C'}$ *in the old version of the model.*

In our framework, every version of the model has a unique label (because there can be more than two versions). We will use this label in the beginning of

source version v
context Customer::firstName
init: self.prev().name.tokenize('\\s+')->first() // $init_{firstName}$

context Customer::lastName
init: self.prev().name.tokenize('\\s+')->last() // $init_{lastName}$

context Purchase::totalPrice
init: self.prev().item->collect(price * quantity)->sum() // $init_{totalPrice}$

Figure 6.3: Integrity constraints for purchase schema

the script to identify the previous version for the script. The calls of *prev* in the script will point to that version.

Figure 6.3 demonstrates the usage of *prev* function for the example from Section 6.1. The first and second constraints have the same structure – the previous version of a `Customer` is reached via *prev*, the value of `name` is splitted by whitespace and the first string becomes the value of `firstName` and the last the value of `lastName`[1]. In the third constraint, the previous version of a `Purchase` is reached again via *prev*, its items are iterated and for each item, a price is computed. Finally, *sum* function computes the total price.

Theoretically, it would be possible to refer to the current version of the model in the expression $init_{\widetilde{A'}}$. However, we chose not to support this so that the adaptation of each document can be one-pass[2].

Similarly as for attributes, OCL allows for initialization of associations between objects, which in the XML domain corresponds to creating subtrees of elements. The skeleton of the syntax is the same, only now we are initializing an association $\widetilde{R'}$.

context $\widetilde{C'}$::$\widetilde{R'}$
init: $init_{\widetilde{R'}}$

However, OCL lacks an expression[3] that returns a new instance of a class (analogy to constructors in procedural object-oriented languages). Therefore we add a new kind of OCL literals – class literals.

Definition 16. ClassLiteralExp *is a new kind of* LiteralExp *[66].* ClassLiteralExp *is written in concrete syntax as*
ClassLiteralExpCS ::=
'new' classNameCS '{' variableDeclarationListCS '}',
where classNameCS *is an identifier of a class and* variableDeclarationListCS *contains initialization of the attributes. The expression returns a new object o of type* classNameCS *the properties of o are initialized according to* variableDeclarationListCS.

We will use class literals to initialize the list of realized and pending purchases from the motivational example in Figure 6.4 uses class literal to create new instances of `Purchase` class.

[1] We expect each person to have exactly two names for simplicity.

[2] This does not limit the 'power' of the adaptation algorithm – no additional semantic information can be obtained by navigating the new version of the document.

[3] At least in version 2.3.1.

We would like to point out that an attempt to simplify the expression to:

parent.prev().purchase->select(status = 'realized')

will be rejected by an OCL compiler, because it is not correct w.r.t. the type system – class `Purchase` in the old version is a different class than `Purchase` in the new version. Association end `RealizedList::purchase` expects instances of class `Purchase` from version $\tilde{v}$, but the expression «parent.prev().purchase» returns instances of `Purchase` from version v. The expression in Fig. 6.4 also does not deal with the association between classes `Purchase` and `Item` (splitted in two `purchase` associations in $\tilde{v}$). One option is to write a separate initialization expression for `purchase` associations. The drawback of such approach is that it would require us to write initialization expression in OCL for every attribute and association in a subtree of some association, which was initialized in OCL. This is not desirable, because we want the algorithm to use the information acquired from the *mapping* in most cases and use OCL only when necessary – when the mapping is annotated. That is why we need a way to signal that we are 'returning back'. For this purpose, we add another function *next*, which is a counterpart of *prev*. Fig. 6.5 demonstrates the usage of *next* function.

Definition 17. *Function* next() *can be called on any instance (object) of class* C' *in the source schema, which is linked to class* $\widetilde{C'}$ *in the target schema via a version link* $(C', \widetilde{C'}) \in \mathcal{VL}$. *The return type of* C'.next() *is then* $\widetilde{C'}$. *Calling* next() *returns the instance of* $\widetilde{C'}$ *in the new version of the model.*

Figure 6.5 demonstrates the usage of *next* function in the expression «$p.next()$». Here, p is of type `Purchase` $\in \mathcal{S}'_c$. The type of the expression «$p.next()$» is, by Def. 17, `Purchase'` $\in \widetilde{\mathcal{S}}'_c$. The value of «$p.next()$» is computed by the host language implementing the adaptation script. In the next section, we will describe, how *next*, *prev* and class literals can be implemented (we will extend translation of the adaptation algorithm (Algorithm 2) from Sec. 4.3 and the translation of OCL constraints from Sec. 5.5).

6.3 Translating Annotations to XPath/XSLT

In this section, we will show how to incorporate the OCL mapping annotations into the adaptation stylesheet. We will use XSLT as the implementation language, as we did in Chpt. 4 and 5.

The heart of both types of annotations – attribute and association initialization – is an OCL expression $init_{\widetilde{X'}}$ (where X' is either an association end or an attribute). The expression $init_{\widetilde{X'}}$ can be an arbitrary OCL expression, which can contain (besides standard constructs introduced in 5) several extension constructs

source version v
context RealizedList::purchase
init: parent.prev().purchase->select(status = 'realized')->
collect(p | **new** Purchase{status = p.status, code = p.code})

Figure 6.4: Initialization of an association

```
source version v
context RealizedList::purchase
init: parent.prev().purchase->select(status = 'realized')->collect(p | p.next())

context PendingList::purchase
init: parent.prev().purchase->select(status = 'pending')->collect(p | p.next())
```

Figure 6.5: Using *next* to signal a return to mapping based adaptation

– functions *prev* and *next* to traverse between the source and target schema and class literals to indicate creating new instances.

6.3.1 Translating Class Literals

Class literals allow us to initialize subtress in $init_{\widetilde{X'}}$. We need to find a corresponding construct for XPath/XSLT. XPath by itself does not allow to create new nodes in expressions. This is only available in XQuery (using node constructors).

In XSLT, new nodes can be created in XSLT instructions which can contain sequence constructors. On such instruction is `xsl:function`, which defines a new XSLT function. Functions defined in a stylesheet can also be called from XPath expression – thus, by calling an XSLT function, we can bypass the limitations of XPath – from an XPath expression, we can call an XSLT function an this function constructs the new elements.

Figure 6.6 contains our *generic constructor* function, which creates a new element with given name and content (the content is passed as an XSLT 3 map item). Generic constructor can be called from an XPath expression, e.g.,: `oclX:genericConstructor('Purchase', ...)`.

6.3.2 Translating *prev* Function

We introduced *prev* function to allow the system designer to declare the semantic relationship between the source and target schema. It can be used in $init_{\widetilde{X'}}$in those positions, where data (information) need to be pulled from the adapted

```
<xsl:function name="oclX:genericConstructor" as="item()*">
 <xsl:param name="element-name" as="xs:string"/>
 <xsl:param name="subelements" as="map(*)" />

 <xsl:element name="{$element-name}">
  <xsl:for-each select="$subelements">
   <xsl:variable name="key" select="map:keys($subelements)" />
   <xsl:element name="{$key}">
    <xsl:sequence select="$subelements($key)" />
   </xsl:element>
  </xsl:for-each>
 </xsl:element>
</xsl:function>
```

Figure 6.6: Generic element constructor

document. This data can be later modified by other parts of the expression $init_{\widetilde{X'}}$.

In Sec. 4.3, we described how for each changed class a top-level *match* template is created in the adaptation XSLT. Inside each template, a node in the source document corresponding to the adapted classs, is accessible as the current node of the template. We will translate the expression $\widetilde{C'}.prev$ as an XPath relative path expression P, where P satisfies the following:

1. P returns the portion of the source XML corresponding to C'.
2. P is relative to the current node of the template where the expression $\widetilde{C'}.prev$ is used.

Thus, the expression «self.prev()» in $init_{firstName}$ is translated as simply as the expression returning the current node: `.` and it returns the XML element `purchase` being adapted. If we wanted to refer to the information from `Purchase` when adapting `Item`, we could write «self.parent.prev()» («self» is of type `Item`, navigation «parent» returns an instance of `Purchase`) and this expression will be translated as `./parent::purchase` or just `..`, because the current node of the template adapting `Item` is `item` and element `purchase` is its parent node.

6.3.3 Translating *next* Function

Counterpart function to *prev* function is *next* function. It can be called on a class C' participating in version link (C', $\widetilde{C'}$). It si used to signal that the transformation from an instance of C' to an instance of $\widetilde{C'}$ not described in the OCL script in the current OCL expression. Because we are using XSLT, the instance of C' should be passed to the corresponding XSLT template. This is achieved by using XSLT `apply-templates` instruction.

We have a similar problem as with the generic constructor wrapping `element` instruction – we need a functionality provided by a certain XSLT instruction, but XSLT instructions can not be called directly from an XPath expression (this time, we want to call `apply-templates` instruction). Again, we have to wrap the instruction in an XSLT function and call this function instead. Figure 6.7 depicts the wrapping function.

Let «E» be some OCL expression. We will translate the expression «E.next()» into XPath expression `oclX:apply-templates(`X_E`)`, where X_E is an XPath expression obtained by translating expression «E».

In our motivational example, *next* is used to adapt class `Purchase` in the expression initializing association end `Purchase`:

context RealizedList::purchase
init: parent.prev().purchase->select(status = 'realized')->collect(p | p.next())

```
<xsl:function name='oclX:apply-templates' as='item()*'>
 <xsl:param name="target" as='item()*'/>
 <xsl:apply-templates select='$target'/>
</xsl:function>
```

Figure 6.7: XSLT `apply-templates` instruction wrapped as a function

```
<xsl:template match='/customer'>
 <customer>
  <xsl:variable name='firstName-new'
    select= 'tokenize(name,'\s+')[1]' />
  <xsl:variable name='lastName-new'
    select= 'tokenize(name,'\s+')[2]' />
  <firstName>
   <xsl:sequence select='$firstName-new'/>
  </firstName>
  <lastName>
   <xsl:sequence select='$lastName-new'/>
  </lastName>
  <xsl:call-template name='RealizedList'/>
  <xsl:call-template name='PendingList'/>
 </customer>
</xsl:template>

<xsl:template name='RealizedList'>
 <realizedPurchases>
  <xsl:variable name='purchase-new'
    select="for $p in purchase[status eq 'realized']
    return oclX:apply-templates($p)"/>
  <xsl:sequence select="$purchase-new"/>
 </realizedPurchases>
</xsl:template>
```

```
<xsl:template name='PendingList'>
 <pendingPurchases>
  <xsl:variable name='purchase-new'
    select=''for $p in purchase[status eq 'pending']
    return oclX:apply-templates($p)''/>
  <xsl:sequence select="$purchase-new"/>
 </pendingPurchases>
</xsl:template>

<xsl:template match='/customer/purchase'>
 <purchase>
  <xsl:apply-templates select='code' />
  <xsl:variable name='totalP-new'
    select='sum(for $i in item
      return $i/price * $i/quantity)' />
  <totalPrice>
   <xsl:sequence select='$totalP-new'/>
  </totalPrice>
  <xsl:apply-templates select='item' />
 </purchase>
</xsl:template>
```

Figure 6.8: Translation of the OCL annotations

The subexpression «p.next()» will be translated into an XPath expression `oclX:apply-templates($p)`. Call to apply templates will ensure that a correct *match* template is triggered when adapting `Purchase`. This way, it is possible to signal from the OCL expression that the structural adaptation algorithm (based on version links) should be called again.

To conclude this section, Figure 6.8 show the updated translation of the templates from the motivational example where OCL annotations are involved[4]. The complete stylesheet can be found in Appendix B.2.

With the combination of structural adaptation algorithm from Chpt. 4, translation of OCL constraints and semantic annotations, the user can specify the mapping between the versions of a document to an arbitrary detail and the generated adaptation script can be more versatile. The limit of the approach is that it can not (in its current state) connect to other data sources during adaptation – all required data must be present in the adapted document. We leave the prospect of connecting to other data sources (external documents, databases, the Internet...) for our future work (see Chpt. 9). Finally, though it is definitely possible to define whole new XML subtrees in the adapted documents using class literals, it is cumbersome. We get back to this problem again when discussing open problems (see Chpt. 9.3).

[4]Translations for `last-name` and `PendingList` are omitted, because they are variations on the previous translations

7. Implementation

We gradually incorporate the results of all our research activities related to our 5-level framework (see Figure 2.1) into an experimental tool *eXolutio* [43]. Fig. 7.1 depicts a screenshot of the application. We will first describe its overall architecture, then we focus on the modules related to this thesis.

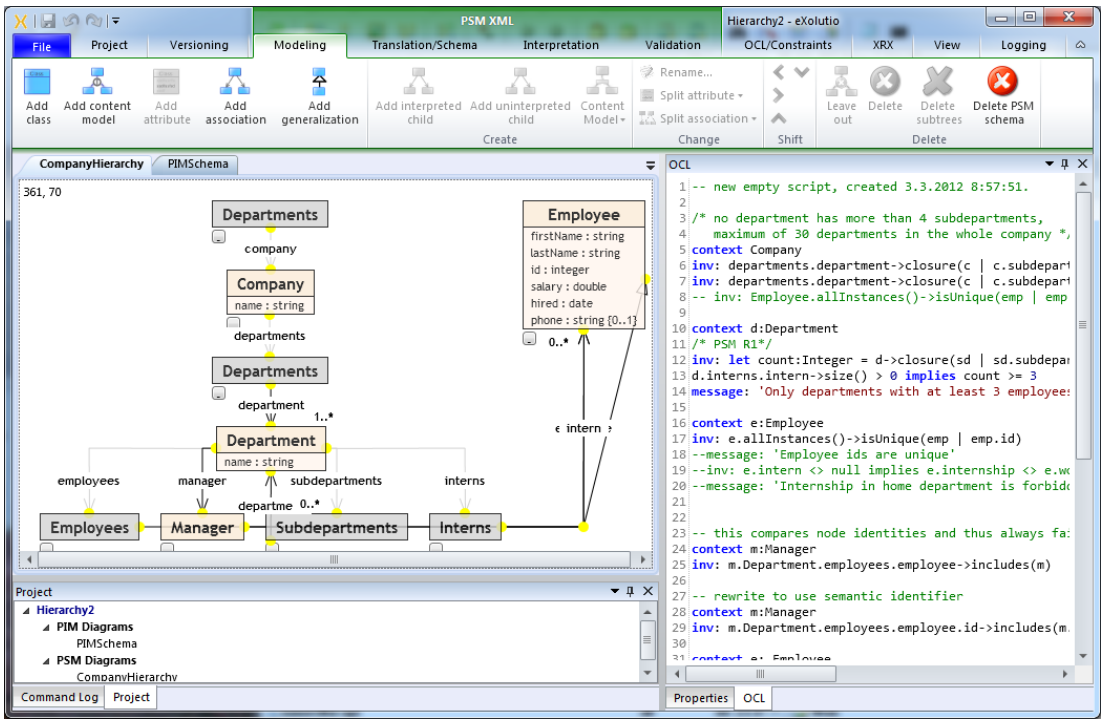


Figure 7.1: *eXolutio*

7.1 Architecture

The architecture of *eXolutio* is based on the well known Model–View–Controller (MVC) design pattern. This means that we hold all the project data in the *model* component. The structure of the model component corresponds to the definitions from Chpt. 3. The model contains one or more versions ($\mathcal{V}$) of the system, each version contains one PIM schema and arbitrary amount of PSM schemas. The model contains the relation of version links ($\mathcal{VL}$). A PIM or PSM schema can contain a set of OCL scripts with integrity constraints. The model does not expose any operations or methods. Modifications to the model are performed through controller.

The *controller* component contains operations of schema evolution. There is a finite set of atomic operations and an extensible set of composite operations, where composite operations are build from other operations (atomic or composite). In [56] we describe the theoretical background for our atomic and composite operations. OCL expressions are edited using an integrated editor. Besides the operations described in [56] (in that paper we do not consider multiple versions of the model), the controller also contains operations concerning versioning, such

as *branch* (see Chpt. 3.3), *create version link* etc. Usually, the user only needs to use *branch* to create new version and the system can manage version links automatically, but manual corrections of version links are allowed as well.

The responsibility of the *view* component is to display the model to the user and update each time the model is modified (each time the controller executes an operation). The connections between individual parts are loose enough so it is possible to, e.g., use multiple visualizations. We have a Windows Presentation Foundation [51] visualization (a desktop application) and a proof-of-concept Silverlight [50] visualization (a browser application). Both visualizations share the same model and the same controller.

7.2 Adaptation

The user can edit the schemas and create new versions (using *branch* operation), modify version links and add OCL annotations in the tool. All versions of the system can be edited.

When the user selects two versions of a PSM schema, he can generate an adaptation script in XSLT. The adaptation script uses XSLT 3.0 [93] and can be executed in any conforming processor. It must be noted that [93] does not have a status of a recommendation yet (it was a working draft at the time of writing this thesis) and may be subject to change. However, the core construct of higher-order functions is shared with XPath/XQuery (which is now a candidate recommendation), is already implemented by several vendors [76, 49, 4, 19] and can be considered quite stable.

The structure and properties of the generated XSLT script are described in Chpt. 4.3, the algorithm was further enhanced in Chpt. 6. The script can be used ad-hoc to adapt individual documents, it can be used in a batch adaptation-process or it can be integrated into a software component (majority of platforms allows to call XSLT in some ways) that adapts documents on demand. When the script is created, it is not dependent on *eXolutio* and to run it does not require any information from the more abstract levels of our framework.

7.3 Validation of Integrity Constraints

We integrated an OCL editor into *eXolutio*, for both PIM and PSM level OCL expressions. Expressions for semantic adaptation (see Chpt. 6) can be added for the evolved schemas. The OCL editor provides standard editing features and syntax highlighting. OCL expressions are processed by a parser created using ANTLR [69].

The tool implements algorithm for suggesting/translation of relevant constraints from PIM to PSM (steps 1.-3.) and translation from OCL (4.) to Schematron schemas. The user may choose between schema-aware and non-schema-aware (which add data conversion for extracting typed values from the XML document) schema and between implementation of iterator expressions using dynamic evaluation or higher-order functions. Translating iterator expressions using dynamic evaluation is described in [31] as an alternative to higher-order functions. Dynamic evaluation is available in some XSLT 2.0 processors [76] as a vendor extension.

The generated Schematron schema can be then used to validate an XML document. XProc pipeline is used to perform the validation. It first executes the transformation steps from standard Schematron pipeline (5.), adds includes for OclX library (6.) and then validates the document (7.) with the resulting XSLT. The pipeline expects the schema (5.) and validated document (8.) on its input ports and writes validation result - a SVRL document - to its output port (9). Again, when the schema is created, it is not dependent on *eXolutio* and can be run using any XProc processor (or using an XSLT processor in several steps).

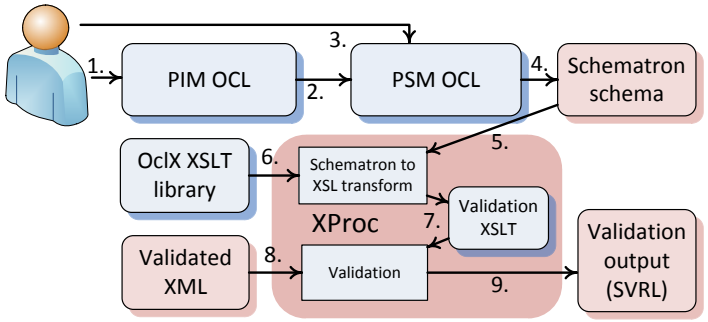


Figure 7.2: Workflow for integrity constraints modeling and validation

OclX library can be downloaded from [42] and used as a standalone library for XSLT development in a functional-programming style.

In Chpt. 5.5.8, we introduce rewritings of certain queries. These can be applied as a part of step the translation step (4.). The selection of rewritings is interactive – the user is offered with a list of subexpressions, at which rewritings can be applied. For each rewritings, a list of options is offered and he can choose applications of rewritings individually.

8. Related Work

In this chapter, we discuss related work in the areas of schema evolution, document adaptation and integrity constraints.

8.1 Schema Evolution and Document Adaptation

Currently there exists a significant number of approaches that detect changes between two schemas of data and output the sequence of edit operations that enable document adaptation [44]. For the goal of determining whether $\mathcal{T}(\mathcal{S}')$ was invalidated, the system must recognize and analyse the differences between $\mathcal{S}'$ and $\widetilde{\mathcal{S}}'$. There are three possible ways to recognize changes:

a) only a single change allowed with immediate propagation,

b) recording of the changes as they are conducted during the design process,

c) comparing the two versions of the schema and looking for changes.

We will briefly describe the consequences of choosing the first trivial approach and then discuss in more detail the other two. Either way has both advantages and disadvantages and the choice between the two significantly influences the capabilities of any adaptation algorithm.

The supported operations can be also variously classified. For instance, paper [78] proposes *migratory* (e.g., movements of elements/attributes), *structural* (e.g., adding/removal of elements/attributes) and *sedentary* (e.g., modifications of simple data types) operations. Classification according to complexity distinguishes *atomic* and *composite* operations.

The changes can also be expressed variously and more or less formally, sometimes using domain-specific languages created for this purpose. For instance in [10] a language called *XSUpdate* is described. In [21] the authors propose the *XSchemaUpdate* language[1].

8.1.1 Incremental Evolution with Immediate Propagation

The basic approach is to allow only a single change between $\mathcal{S}'$ and $\widetilde{\mathcal{S}}'$ which is immediately propagated to $\mathcal{T}(\mathcal{S}')$. In particular, the tool offers a finite set of evolution primitives, the user selects one and as the schema is evolved, the documents are adapted right away. As this approach does not support conducting multiple operations in one evolution cycle, adaptation is not effective (with more operations, the set $\mathcal{T}(\mathcal{S}')$ will be adapted over and over, each time with only a small local change). Moreover, tools that choose this approach offer a limited set of primitives, which results in the necessity of using more primitives for one logical operation (e.g. 'rename element' requires 'remove element' and 'add element'). This results in unwanted loss of data. Since each primitive is propagated immediately, such an approach can never meet the requirement for separation of the schema evolution and document adaptation phases (they are interleaved).

[1] Both *XSUpdate* and *XSchemaUpdate* are not to be confused with W3C XQuery Update Facility [91] language

System *X-Evolution* [24] is built upon a graphical editor for creating schemas in the XML Schema language. Each single evolution operation executed upon the schema is immediately propagated to valid documents. Backward compatible operations are identified (and note propagated). Backward compatibility is decided particularly by marking each evolution primitive as backward-(in)compatible.

The set of supported operations is limited to insertions and deletions, which need to be used for all more complex operations (like move, adding a wrapping element etc.). A part of the set of available operations is rather technical (switching between a local and global definition of a type etc.). It considers only elements without attributes and recognizes choice/sequence/set content models. *X-Evolution* does not use any conceptual model for the schemas.

In [21] G. Guerrini and M. Mesiti further describe the *XSchemaUpdate language*. Using statements in this language, a new content can be created with non-default values.

The system is implemented as a schema visualization tool, where the user can select a construct which (s)he wants to evolve and then select the desired evolution primitive from the set of available primitives. Revalidation of $\mathcal{T}(\mathcal{S}')$ is triggered right after an evolution primitive is selected.

XEM [77] is another immediate propagation approach to manage schema evolution, but this time, DTD is used as a schema language. It deals both with changes in DTD and XML documents. Both DTD and instance XML documents are represented in the system as directed acyclic graphs and the evolution primitives are defined as operations on these graphs. For each DTD altering primitive a resulting data change is defined in terms of the primitives altering instance documents.

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; however, there are primitives for addition and removal, but none for moving or renaming. Any change in DTD can thus be expressed via the proposed primitives, but when these are propagated to the valid XML documents, they lead to removing a significant part of the XML document and recreating it again. This applies even for small, local changes. E.g., when an element needs to be renamed, it must be removed first (which can only be done after its subtree is removed) and then added under new name. In this process the structure is created properly by the algorithm, but the data is lost.

Since *XEM* works with DTDs, its capabilities are restricted (e.g., no support for set content model, limited support for cardinalities etc.). A conceptual model is not utilized.

There are other papers which deal with propagation of a single change expressed in DTD [1, 16] or XSD [78, 10] to respective XML documents. There also exists an opposite approach that enables one to evolve XML documents and propagate the changes to their XML schema [7].

Problems of Immediate Propagation

Immediate propagation approaches carry certain drawbacks and disadvantages. The most prominent is the mingling of schema evolution and document adaptation (we discussed this issue in Chpt. 1.3). As every change in the schema is propagated immediately to all documents, the approach can be very ineffective

when the set $\mathcal{T}(\mathcal{S}')$ is large – after each operation, the user has to wait until all documents are adapted, before he can continue with another operation. If the user conducts operations that cancel each other or a conducts several operations, that are backwards compatible only as a whole, the tool cannot optimize adaptation. Version comparison and change recording approaches target these drawbacks.

8.1.2 Version Comparison and Change Detection

In general, with approaches that compare schemas, the user can edit both schemas independently until (s)he is satisfied with them. The change detection algorithm then takes the two schemas as an input and compares them. The result of the comparison is a list of differences between the schemas. The characteristics of the comparison approach are as follows:

- The approach allows for clear separation of schema evolution phase and document adaptation phase. Batch adaptation is also possible.
- Some type of changes are ambiguous and cannot be distinguished without additional information or user's decision (*rename* vs. *add* & *remove*, *move* vs. *add* & *remove*), methods for mapping discovery are necessary.
- There is no need to look for redundancies; the set of changes is always minimal.
- Both old version $\mathcal{S}'$ and new version $\widetilde{\mathcal{S}}'$ can be edited without limitations. The system may contain an arbitrary amount of versions, any pair of versions can be compared. This can increase efficiency, e.g., when a document valid against version S'_1 is revalidated against S'_3 – the script obtained from comparing S'_1 and S'_3 can be more efficient than the concatenation of the scripts for transformations $S'_1 \to S'_2$ and $S'_2 \to S'_3$.
- The process of schema evolution can be arbitrarily stopped and resumed.
- The reversed 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 and serve as an input to the change detection algorithm.

Change detection of two given versions of data is a key part of, e.g., data integration, versioning, similarity evaluation, etc. [73] In all the cases we are interested in the sequence of edit operations, which is further used for mapping purposes, evaluation of the degree of difference etc. At the level of XML documents we can restrict ourselves to detecting changes between trees, either ordered [13, 40] or unordered [11, 95]. Since the problem is proven to be NP-hard [13], various heuristics reflecting the respective application are often incorporated, as well as optimization strategies for processing large documents and gaining better results using relational databases [40, 39], XPath queries [71], tuning steps [38] etc.

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

shows that the approaches for XML data cannot be directly applied for XML schemas due to the mentioned semantics of nodes is proposed in [41]. It exploits a heuristics based on MD5 hashing. There exists also an approach that detects changes among XML schemas for the purpose of evaluation of their similarity on the basis of classical tree-edit distance [97].

The algorithm in [37] starts at roots of the source and target XML schema and continues recursively (the routine attempts to match sets of children of two already matched nodes). The mapping is "best-effort" and partial, hence the produced XSLT script does not guarantee correct revalidation (the output document will not be valid against the target schema) – the user is expected to adjust it. The commercial tool [2] offers semi-automatic schema comparison (of XML schemas, i.e., at the logical level) and subsequent creation of an adaptation script, but the task of resolving ambiguities inherent in schema comparison approaches is left up to the user.

8.1.3 Recording Changes

A system that records changes has the advantage of knowing the sequence of operations that were performed. However, the sequence does not have to be optimal, because the user could reach the result using various more or less reasonable sequences of operations. Here is the outline of the key characteristics of recording approach:

- The approach allows for clear separation of schema evolution phase and document adaptation phase. Batch adaptation is also possible.
- Recorded set provides enough information to propagate changes in the schema to the documents, there are no ambiguities or possible misinterpretations of operations.
- The recorded set may contain redundancies (repeated changes in the same location etc.), but it could be normalized to eliminate them.
- Once the evolution process is started, the old version $\mathcal{S}'$ cannot be easily changed.
- A user may want to interrupt his/her work at some point and continue in another session. The sequence of recorded changes would have to be stored and recording resumed later.
- When the user wants to retrieve the sequence for reverse process, (s)he will have to either start with the new version $\widetilde{\mathcal{S}}'$ and record the operations needed to go back to the old version $\mathcal{S}'$ again, or the system will have to be able to create an inverse sequence for each sequence of operations.
- When the evolved schema comes from an outer source, the sequence of operation changes cannot be retrieved directly; the user must start with his/her old version $\mathcal{S}'$ and manually adjust it to match the new schema $\widetilde{\mathcal{S}}'$.

System *CoDEX* (*Conceptual Design and Evolution of XML Schemas*) [33] is an example of an approach to schema evolution using the true recording approach. The changes made in the visualization of the schema are logged and when the evolution process is finished, the resulting sequence of changes is normalized (using static, but extensible set of rewriting rules) and then performed in the XML schema and respective XML documents. The approach also recognizes

backward-compatible changes.

The visualised model used by CoDEX is not a conceptual model, even though it hides some technical details of XML schema languages.

8.1.4 Other Approaches

In [18] the authors propose an approach for expressing changes at the level of UML classes and their propagation to respective XML schemas and XML documents with the emphasis on *traceability*, i.e., preserving the links between the levels. This is probably the closest system to our basic idea of five-level evolution framework [62]; however, the authors consider only its part. The authors of [18] do not consider operation move or other more complex operations (adding, deleting, renaming and changing a property to a class is supported) and the framework provides directly the output documents, not the set of operations in some standard syntax, that might be further processed with regard to the respective operation.

In [72] the authors propose an algorithm for *incremental schema validation*, i.e., checking validity of an XML document with regard to a modified schema, whereas the aim is not to check the whole document, but only necessary parts. For this purpose, the old and the new version of the schema are analysed and an *intersection automaton* is built from the two versions of the schemas. A modification of the algorithm also enables to efficiently check validity of a modified XML document against the new version of XML schema.

8.1.5 Summary

We have identified several ways of how XML document adaptation is approached in the existing work. All approaches to our best knowledge work with predefined (and thus limited) set of operations (evolution primitives) in the phase of schema evolution. When the schema designer needs to make a change which does not fit to the predefined set of changes, he must use several operations to compose the change. This may be done safely on the platform-specific or logical (schema) level, but when the change is propagated to the extensional level (adapted documents), it may result in data loss or unexpected outcomes. The adaptation script thus require some kind of manual correction in the more complex cases, usually using the implementation language of the script (with the exception of [21], which introduces a language specifically designed for defining updates). Our approach of OCL annotations (see Chpt. 6) solves more complex changes by formally describing an arbitrary mapping on the platform-independent or platform-specific level, without being limited to a predefined set of operations/scenarios.

8.2 Integrity Constraints in Schemas, OCL

Existing academic works, e.g., [96, 3], in the area of integrity constraints for XML focuses mainly on the fundamental integrity constraints (ICs) known from relational databases – keys, unique constraints, foreign keys and inverse constraints – and their mathematical properties, such as decidability, consistency, tractability

(with separate results for one-attribute vs. multi-attribute and relative vs. absolute keys). Paper [20] studies the problem of consistency of a set of ICs and a DTD, i.e., whether a finite XML document, which is valid against a given DTD and satisfies the ICs, exists. The problem is proven to be undecidable for the most general class o ICs, but for the class of unary keys and foreign keys, it is proven NP-complete.

Authors of [8] propose validation of ICs in XML using attribute grammars and automatons. Their path language is less expressive than XPath or OCL, but thanks to this limitation, the validation can be achieved in one pass of the XML document in linear time. The aim of our approach was to support the largest subset of OCL as possible and translate OCL constraints into XPath constraints with similar structure.

ICs spanning several XML documents were studied in [64]. The project converts ICs into XLink [83] links and the consistency can be checked through verifying the validity of the generated links.

Several approaches for modeling XML using UML were proposed [15, 18, 75], but they deal mainly with modeling the structure of the schemas, without debating the integrity constraints present in the model. OCL and UML and related technologies are being researched [27] at Technische Universität Dresden, which is also the coordinator of the leading open-source implementation – Dresden OCL [79]. Dresden OCL research was mainly targeting relational databases platform [17]. A generic framework for generating for translation OCL expressions into other expression languages was proposed in [26]. It mentions 2 applications: OCL → SQL translation and also OCL → XQuery [6]. The OCL → SQL patterns are based on [17], OCL → XQuery on [22]. The expression are translated into the target language via patterns. It expects much tighter mapping between UML model and XML schema (unlike PIM/PSM schemas used in our approach, it does not consider regular properties of schemas). The paper does not describe how to translate general iterator expressions, as we did in Sec. 5.5. The authors support constructs corresponding to projections, cartesian products and restrictions in the expressions (omitting the general iteration and closures facilities).

Authors of [22] examine the fundamental similarities of the two expression languages – OCL and XQuery. They propose a mapping from XQuery queries to OCL constraints (bottom-up approach). They show how the parts of elementary XQuery expressions can be mapped to OCL constructs, but they do not elaborate on translating definitions of and references to (local) variables, which would be interesting for queries with multiple variables (such queries correspond to more complex OCL iterator expressions, which are not mentioned in the paper, and which we translate using XSLT higher-order functions). In consequence, the full expressive power of OCL is not harnessed (for more on expressive power of OCL, see [48]).

CONSIDER: Neco vic o IC

9. Open Problems and Future Work

In this chapter we describe possible enhancements and improvements of our approach.

9.1 Evolution of Constraints

In our paper [56], we have proposed a framework of atomic and composite operations for schema evolution. The operations work at the platform-independent and platform-specific levels (see Fig 2.1) and allow for coherent evolution of a family of PSM schemas. Operations proposed in [56] work with PIM and PSM schemas as defined by Def. 1 and 2 (XML schemas of the extensional level are generated automatically from the PSM schemas). The extension of the framework in [35] also considers inheritance in both PIM and PSM. The key principle of the operations is their *propagation* – a change on one level is propagated to the neighbouring levels and the system as a whole stays consistent. Propagation is defined for atomic operations, composite operations are created strictly by combining multiple atomic operations (thus when a new composite operation is created, the propagation is defined by the framework).

In this work, we include expression languages (OCL) both for PIM and PSM. However, PIM and PSM expressions will not be resilient to schema evolution in the majority of cases (e.g., renaming an attribute would invalidate all expressions referencing that attribute). To support expressions-aware evolution, the atomic schema evolution operations described in [56] would have to be extended to propagate not only to the neighbouring levels, but also to the expressions at the same level. A tightly related topic of query adaptation was studied in [70].

Another interesting topic is the backwards compatibility for schemas with constraints. We studied backwards compatibility for structural schemas in 4.1.2 and introduced conditions (NI-predicates), which guarantee backwards compatibility for schemas without constraints. However, when both the original and the evolved schema have their own integrity constraints, NI-predicates do not guarantee that the integrity constraints in the evolved schema will not be violated.

9.2 Version Links for Imported Schemas

As we have mentioned, before our adaptation algorithm can be used, it requires the relation $\mathcal{VL}$ to be defined (i.e., version links joining the previous and the evolved version of each construct). If the schemas are evolved using our tool [43], the tool can manage the relation automatically in the majority of cases (and the user can alter the relation manually when necessary).

If we want to use our adaptation algorithm for schemas managed elsewhere (e.g., by a third party or a standardization authority), we can import both versions of the schema into the tool. When the schemas are imported, the relation $\mathcal{VL}$ will be empty and must be created manually by the user. This is a time-consuming process.

It would be useful to extend the system with a heuristic that could create a

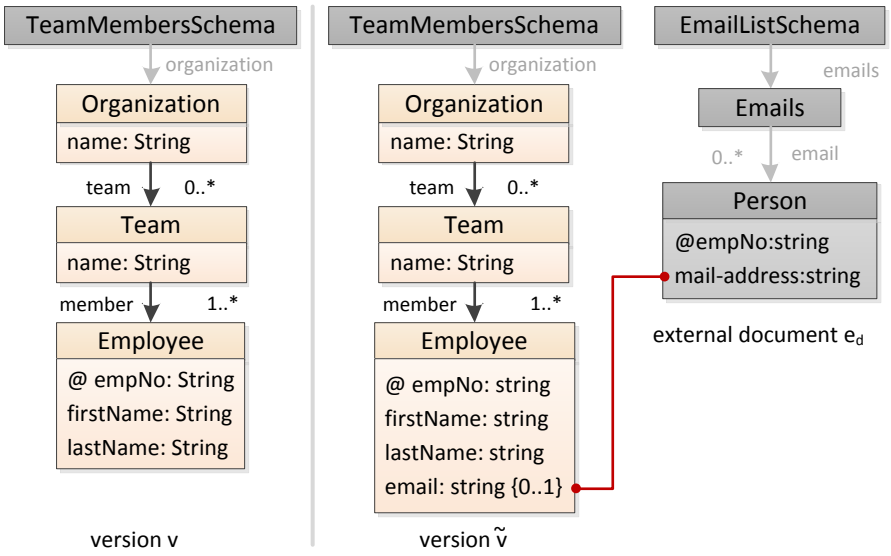


Figure 9.1: Referencing external content in an adapted schema

larger part of $\mathcal{VL}$ automatically and ask for user input only in the unresolved cases. There exists a lot of methods for finding similarities and patterns among two XML schemas (a survey can be found in [57]). Outcomes of these methods can be used when searching for an algorithm for finding similarities between two PSM schemas/two versions of a PSM schema and thus discovering version links for $\mathcal{VL}$.

9.3 Content Templates, External Content

To date, our document adaptation approach is able to deal with changes that modify the structure and data present in the document. However, scenarios, where new content must be added to the adapted documents, is frequent. This can be realized either by modifying the generated adaptation script or by running some kind of update query on the adapted document (after the adaptation script is applied). Currently, our approach does not provide any support for these scenarios.

In Chpt. 4.2.6, we indicated approaches how to handle adding content during adaptation. Besides the basic methods of using default values or letting the user to supply the content manually, other methods can be utilized, one we have proposed in Chpt. 6 – OCL initialization expressions. Using initialization expression, it is possible to define derived values of attributes and also whole XML subtrees (with the introduced class literals). However, defining of complex subtrees using OCL class literals is possible, but cumbersome. We want to address this issue by introducing some kind of templating. The idea is to let the user define a template and then specify (using expressions) how the dynamic parts of the template are filled with data. We are considering two options of how templates will be defined: 1) as "raw" XML documents/fragments or 2) at the PSM level, either as a separate PSM schema or as a part of the evolved schema. The first approach is more straightforward, the second one fits better into the overall architecture of our framework.

When the required data is already present in the system, but not in the adapted documents themselves (e.g., in a database, in other XML documents, available through some service interface), these sources can be used to retrieve the data during adaptation. In our future work, we want to extend our framework in such a way that it will be possible to define the relations between the source XML document, the adapted XML document and other (external) documents/data sources. This information should be defined either on the platform-independent or platform-specific level and the adaptation script should take it into consideration.

To be able to model this, the external data should be mapped to the PIM as another kind of PSM (the system can support many types of PSMs, not only PSMs for XML formats, but also for relational database schemas etc.). The mapping to the external sources (other PSMs) can utilize OCL expressions (similarly as we did in Chpt. 6).

Figure 9.1 depicts an example of an adaptation scenario, where new content should be added in the adapted document (because new attribute `Employee.email` was added in the evolved schema). There is no initialization or default value provided for the attribute and because the attribute is optional, the adaptation script will not create it in the adapted documents.

The value of `email` should be retrieved from an external document e_d containing the list of emails of all the employees in the organization. This information could be described using OCL annotation from Fig. 9.2. The *import document* clause at the beginning defines an external source (an XML document with a PSM schema) and makes the external document accessible through variable «*ed*». The initialization expression uses a modification of standard OCL *let expression* (*LetExp*, see Chpt. 5.5.1) with added *in external source* part, which declares, that the body expression should be evaluated in the external document «*ed*».

```
source version v
import document ed at emails.xml as EmailListSchema

context Employee::email
init:
let e:string = self.empNo in external source ed
  Email.allInstances()->select(m | m.empNo = e).mail-address

<xsl:variable name='ed' select=''doc('emails.xml')/*'' />
...
<xsl:template match='/organization/team/member'>
 <member>
  <xsl:apply-templates='@empNo | firstName | lastName' />
  <xsl:variable name='email-new'
    select='let $e := $self/@empNo return
            $ed//email[@empNo = $e]/mail-address' />
  <email>
   <xsl:sequence select='$email-new' />
  </email>
 </member>
</xsl:template>
```

Figure 9.2: Initialization expression referencing an external document and its translation into XSLT (excerpt)

The translation into XSLT is also depicted in Fig. 9.2. A global variable `ed` corresponding to the *import document* clause is declared at the top of the adaptation script. This variable is used in the translation of the *let expression* from the initialization expression.

The example in Fig. 9.2 is a straightforward example, where the definition of the mapping is up to the user (the mapping is defined explicitly using OCL, the external document is not mapped to the PIM). A more advance approach would be to infer the OCL expression from the mapping between PIM and the external document(s). Also, the example uses external XML document which can be queried by XPath/XSLT. Other possibilities are to retrieve data by an SQL from a relational database, a SPARQL [88] query from an RDF [84] triple store etc.

9.4 Full XPath Axes Support in PSM OCL, Enhancments of Translation Algorithms

As opposed to OCL, XPath provides a wide range of axes to the user. In Chpt. 5.2, we extended OCL language for PSM and added constructs corresponding to *parent* and *child* axes. However, a user familiar with XPath might find support for only these two axes limiting. It is true that some PSM OCL expressions might become more straightforward, if constructs corresponding to axes such as *descendant*, *following* etc. were available. This is certainly a field for further improvement.

Finally, algorithms presented in Chpt. 5.4 restrain the classes of expressions, most importantly, they do not consider navigation applied on a result of a general function. In such cases, the subexpression must be translated from the PIM to the PSM manually. This step

Conclusion

Document adaptation

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

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

- We exploit the idea of a conceptual model of XML schemas and, hence, the user is provided with a user-friendly tool for expressing changes. We have built the algorithm upon a general schema model, which is proven [61] to be equivalent to regular tree grammars (which form a theoretical background to schema languages used in practice, such as DTD, XML Schema and RELAX NG).
- Our approach is integrated within a five-level evolution framework where the two conceptual levels – platform-independent and platform-specific – together with the versioning support enable to model multiple schema versions at once.
- We overcame the problem inherent in all approaches comparing/mapping two versions of schemas – the need to resolve ambiguities when interpreting changes – via introducing the version links. Each construct in the model is then correctly connected with its counterpart constructs in all other versions, where the construct exists. Adding the version links allowed us to define changes that can occur between two versions v and $\tilde{v}$ of a schema and detects these changes algorithmically.
- Our approach outputs an XSLT script that adapts the modified XML documents with regard to a new version of a schema. The adapted document preservers semantic meaning of the constructs due to utilizing the version links.
- The user works with and evolves a conceptual model of a schema and the mappings are defined for the conceptual model as well. This is a significant improvement compared to working directly with often lengthy and hard to read XML schemas.

In Chpt. 1.3, we proposed requirements for an adaptation framework and the resulting script. To conclude, we examine to what degree our approach meets these requirements.

Set of supported operations It is crucial for an approach to support a rich set of schema evolution operations – The approach should be able to cover a transition of the source schema to an arbitrary schema. This can be achieved plainly by removing and recreating the modified parts of the schema, but the

better solution is to achieve the transition by updates small and local. For this purpose, the framework must provide operations to update all properties of all constructs. Our framework uses a set of atomic schema evolution operations proposed in [56, 35]; atomic operations can be further combined into composed operations. The atomic operations allow to create and remove any supported construct and update all the properties of all constructs.

The two versions of schemas are compared and changes are detected in the schemas. The set of recognized changes, proposed in Chpt.4.1, again, allows to detect added and removed constructs and changes in all properties of all types of constructs. For every type of change, we proposed possible validation in Chpt. 4.2. Our implementation selects one of the alternatives of validation, which can be modified by using OCL annotations introduced in Chpt. 6.

The user can not extend the set of recognized changes, but he can override how a particular instance of a change is adapted, again using OCL annotations. This way, particular adaptation scenarios can be modified.

The generated adaptation script is using standard XSLT, which the user can also alter manually after it is created, before it is applied.

Separation of schema evolution from document adaptation The two phases are separated completely in our framework. The user can edit the old and the new version independently (together with an arbitrary amount of versions). Document adaptation script can be generated for any two versions of the schema, in both directions and repeatedly.

Normalization before propagation Normalization is not performed in our approach, because it makes sense only for approaches where schema evolution primitives map directly to document adaptation update actions. In our approach, schema is adapted as a whole, not change by change.

References to content The approach allows to reference the content of the document using OCL expressions in OCL annotations. Thanks to that, it is possible to steer the adaptation algorithm according to the contents of individual documents (e.g., to introduce if/then/else and choices testing the actual contents of the document into the adaptation script).

Resolving ambiguities Structural ambiguities in our approached are resolved using version links introduced in Chpt. 3.3. E.g., a moved attribute is connected by a version link to an attribute in the old version and that is how the algorithm can decide that the attribute was indeed moved and it is not a new attribute. This requires the set of version links to be maintained – this is done by the tool in the background and the user must intervene only when he wants to manually change the version links. E.g., when an attribute was moved, but the used *add attribbute* and *remove attribute* instead of *move attribute*. In this case, the link must be created manually.

Advanced content generation Our approach creates *default instances* (minimal subtrees, attributes with default values) where content should be generated

during adaptation. If this is not sufficient and the content to be created can be derived entirely from the adapted document, the user can again use OCL initialization expressions and *class literals*, as proposed in Chpt. 6. It must be noted that using OCL to generate complex subtrees is cumbersome and we leave the improvements of this scenario for our future work (see Chpt. 9.3. When the generated content can not be derived entirely from the adapted document, but some data need to be retrieved from another data source, OCL initialization expressions, as proposed in Chpt. 6, are not powerful enough. Utilizing external data sources is also a direction of our future work Chpt. 9.3.

Other aspects Our approach has a formal background, but this formalism is not concealed from the user, who can work with a graphical notation (UML diagrams) in a CASE tool [43].

The approach is language independent at the PIM and PSM level, the schemas, expression/queries and transformations generated by the approach are using standard languages of the XML stack (XSD, Relax NG, Schematron, XPath, XSLT).

It must be noted that when the user wants to utilize integrity constraints and semantic annotations, he can do so using OCL at the PIM and PSM level together with UML class diagrams. We use only a subset of UML which is easy to learn and understand, but OCL is a full-fledged expression language and less well established among XML developers. However, thanks to its design, it allows to reuse expressions in different platforms (they can be translated to Java expressions, SQL queries or XPath, see 8.2). Systems that use several implementation languages and technologies (i.e., Java for business logic, SQL and/or XML for storage and XML for data exchange) would benefit most from our approach.

Our approach is efficient, because the phase of adaptation is clearly separated from schema evolution. We do not require to adapt after each change in a schema, but only when the whole schema is evolved (and all evolution steps are consolidated). We are able to decide, whether changes made in the schema are backwards-compatible or not. The generated adaptation script is one-pass and skips those portions of adapted documents that were not changed. Its complexity grows proportionally to the number of changes in the schema, not to the size of the schema itself.

Expressions and Integrity Constraints

The second aim of this work was to extend the framework with support for expressions. The motivation came from the adaptation scenarios, which can not be described solely by mapping between structural schemas, but we managed to integrate model expressions into two fundamental applications of our framework – schema design and document adaptation.

With expressions support, the user can model not only grammar based schemas, but is also given finer control over the contents of the document by the possibility to also model rule-based Schematron schemas. We have shown how to write OCL constraints for XML and how to obtain them from existing constraints defined over the platform-independent model. We proposed a translation from OCL into XML (Schematron) schemas, which can be used directly for validation.

Using our tool, it is possible to reuse the definition of an integrity constraint from the conceptual level easily to generate actual verification/validation code for all the places where the constraint is relevant. Thanks to automated translation process, the system designer may focus on the work at the conceptual level and make the model as accurate as possible using UML and OCL. This may be of more appeal to him than working with platform-specific languages (Schematron and XPath). Also, when the constraints are defined using OCL over a UML model it is easier to maintain them when the schema changes. This is allowed because of the tight connection between the two languages. When the model changes, it is very easy to find out which constraints were influenced by the change. A keen tool may even suggest/perform corrections automatically.

When there are several XML formats where a certain constraint should be checked, the user can define the constraint only at the abstract level, without being concerned about the structure (arrangement) of a concrete schema. Our algorithm will then help him to find the concrete XML schemas, where the constraint is applicable. Then the user can translate the constraint automatically to a form which can be checked/evaluated in the concrete schema. When the structure of a specific schema changes, the constraint just needs to be re-translated.

Last, but not least, constraints described at the PIM level are often much clearer and easier to understand (even for a non-technical person) than their translation to XML schemas. In document adaptation, expressions can be used to make the mappings between structures more precise and to compute contents of adapted documents from the source data. We have also proposed several possible enhancements which could benefit greatly from the availability of expressions in the model.

Bibliography

[1] L. Al-Jadir and F. El-Moukaddem. Once Upon a Time a DTD Evolved into Another DTD... In *Object-Oriented Information Systems*, pages 3–17, Berlin, Heidelberg, 2003. Springer.

[2] Altova. DiffDog – XML Aware Diff/Merge Tool. `http://www.altova.com/diffdog/`.

[3] M. Arenas, W. Fan, and L. Libkin. On the complexity of verifying consistency of xml specifications. *SIAM J. Comput.*, 38:841–880, June 2008.

[4] BaseX. BaseX – The XML Database. `http://www.basex.org/`.

[5] P. V. Biron and A. Malhotra. *XML Schema Part 2: Datatypes (Second Edition)*. W3C, 2004.

[6] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. *XQuery 1.0: An XML Query Language*. W3C, 2007.

[7] B. Bouchou, D. Duarte, M. H. F. Alves, D. Laurent, and M. A. Musicante. Schema Evolution for XML: A Consistency-Preserving Approach. In *Mathematical Foundations of Computer Science*, pages 876–888, Prague, Czech Republic, 2004.

[8] B. Bouchou, M. H. Ferrari, and M. A. V. Lima. Attribute grammar for XML integrity constraint validation. In *Proceedings of the 22nd international conference on Database and expert systems applications - Volume Part I*, DEXA'11, pages 94–109, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C, 2008.

[10] F. Cavalieri. EXup: an Engine for the Evolution of XML Schemas and Associated Documents. In *EDBT '10: Proc. of the 2010 EDBT/ICDT Workshops*, pages 1–10, New York, NY, USA, 2010. ACM.

[11] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In J. Peckham, editor, *SIGMOD Conference*, pages 26–37. ACM Press, 1997.

[12] J. Clark and M. Makoto. *RELAX NG Specification*. Oasis, December 2001. `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`.

[13] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In *ICDE*, pages 41–52. IEEE Computer Society, 2002.

[14] E. F. Codd. Relational completeness of data base sublanguages. *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.

[15] R. Conrad, D. Scheffner, and J. Christoph Freytag. XML Conceptual Modeling Using UML. In *Conceptual Modeling – ER 2000*, volume 1920 of *Lecture Notes in Computer Science*, pages 291–307. Springer Berlin / Heidelberg, 2000.

[16] S. V. Coox. Axiomatization of the Evolution of XML Database Schema. *Program. Comput. Softw.*, 29(3):140–146, 2003.

[17] B. Demuth and H. Hussmann. Using UML/OCL constraints for relational database design. In *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*, UML'99, pages 598–613, Berlin, Heidelberg, 1999. Springer-Verlag.

[18] E. Dominguez, J. Lloret, A. Perez, Beatriz Rodriguez, A. L. Rubio, and M. A. Zapata. Evolution of XML schemas and documents from stereotyped UML class models: A traceable approach. *Inf. Softw. Technol.*, 53:34–50, January 2011.

[19] eXist db. eXist – Open Source Native XML Database. `http://www.exist-db.org/`.

[20] W. Fan and L. Libkin. On xml integrity constraints in the presence of dtds. *J. ACM*, 49(3):368–406, May 2002.

[21] G. Guerrini, M. Mesiti. XML Schema Evolution and Versioning: Current Approaches and Future Trends. In E. Pardede, editor, *Open and Novel Issues in XML Database Applications: Future Directions and Advanced Technologies*, pages 66–87. Idea Group Publishing, April 2009.

[22] A. Gaafar and S. Sakr. Towards a Framework for Mapping Between UML/OCL and XML/XQuery. In *UML*, pages 241–259, 2004.

[23] O. M. Group. UML 2.1.2 Specification, 2007. `http://www.omg.org/spec/UML/2.1.2/`.

[24] G. Guerrini, M. Mesiti, and M. A. Sorrenti. XML Schema Evolution: Incremental Validation and Efficient Document Adaptation. In *XSym*, volume 4704 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2007.

[25] K. Hamilton and L. Wood. Schematron in the context of the clinical document architecture (cda). *Balisage Series on Markup Technologies*, 8, 2012.

[26] F. Heidenreich, C. Wende, and B. Demuth. A framework for generating query language code from ocl invariants. *ECEASST*, 9, 2008.

[27] H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting ocl. In *UML: advancing the standard*, UML'00, pages 278–293. Springer-Verlag, 2000.

[28] 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`.

[29] ISO/EIC. *Information Technology Document Schema Definition Languages (DSDL) Part 3: Rule-based Validation Schematron. ISO/IEC 19757-3*, feb 2006.

[30] ISO/EIC. *Information Technology Document Schema Definition Languages (DSDL) Part 4: Namespace-based Validation Dispatching Language (NVDL). ISO/IEC 19757-3*, feb 2006.

[31] Jakub Malý and Martin Nečaský. Utilizing new capabilities of XML languages to verify OCL constraints. In *Proceedings of Balisage: The Markup Conference 2012*, Balisage Series on Markup Technologies. Mulberry Technologies, 2012.

[32] M. Kay. *XSLT 2.0 and XPath 2.0 4th Edition*. Wrox, 2008.

[33] M. Klettke. Conceptual XML Schema Evolution — The CoDEX Approach for Design and Redesign. In *Workshop Proceedings Datenbanksysteme in Business, Technologie und Web (BTW 2007)*, pages 53–63, Aachen, Germany, March 2007.

[34] J. Klímek and M. Nečaský. Semi-automatic Integration of Web Service Interfaces. *Web Services, IEEE International Conference on*, 0:307–314, 2010.

[35] J. Klímek and M. Nečaský. Formal Evolution of XML Schemas with Inheritance. In *Proceedings of International Conference on Web Services 2012*. IEEE Computer Society, 2012.

[36] J. Klímek and M. Nečaský. Using Schematron as Schema Language in Conceptual Modeling for XML . In *Proceedings of Asia-Pacific Conference on Conceptual Modelling 2013*. IEEE Computer Society, 2013.

[37] M. Kwietniewski, J. Gryz, S. Hazlewood, and P. Van Run. Transforming xml documents as schemas evolve. *Proc. VLDB Endow.*, 3:1577–1580, September 2010.

[38] S. Lee and D. Kim. X-Tree Diff+: Efficient Change Detection Algorithm in XML Documents. In E. Sha, S.-K. Han, C.-Z. Xu, M.-H. Kim, L. Yang, and B. Xiao, editors, *Embedded and Ubiquitous Computing*, volume 4096 of *Lecture Notes in Computer Science*, pages 1037–1046. Springer Berlin / Heidelberg, 2006.

[39] E. Leonardi and S. S. Bhowmick. Xandy: A scalable change detection technique for ordered XML documents using relational databases. *Data Knowl. Eng.*, 59(2):476–507, 2006.

[40] E. Leonardi and S. S. Bhowmick. XANADUE: a system for detecting changes to XML data in tree-unaware relational databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1137–1140, New York, NY, USA, 2007. ACM.

[41] E. Leonardi, T. T. Hoai, S. S. Bhowmick, and S. K. Madria. DTD-Diff: A change detection algorithm for DTDs. *Data Knowl. Eng.*, 61(2):384–402, 2007.

[42] J. Malý. OclX function library, 2012. `https://github.com/j-maly/OclX/`.

[43] J. Malý, J. Klímek, and M. Nečaský. eXolutio project. `http://exolutio.com`.

[44] J. Malý, I. Mlýnková, and M. Nečaský. On XML Document Transformations as Schema Evolves – A Survey of Current Approaches. *ISD 2011*, 2011.

[45] J. Malý, I. Mlýnková, and M. Nečaský. XML Data Transformations as Schema Evolves. In *ADBIS '11: Proc. of the 15th Advances in Databases and Information Systems*, Vienna, Austria, 2011. Springer-Verlag.

[46] J. Malý, I. Mlýnková, and M. Nečaský. Efficient adaptation of XML data using a conceptual model. *Information Systems Frontiers*, 2012.

[47] J. Malý and M. Nečaský. When Grammars do not Suffice: Data and Content Integrity Constraints Verification in XML through a Conceptual Model. In *Proceedings of Asia-Pacific Conference on Conceptual Modelling 2013*. CRPIT, 2013.

[48] L. Mandel and M. Cengarle. On the Expressive Power of OCL. In *FM'99 – Formal Methods*, volume 1708 of *LNCS*. Springer-Verlag, 1999.

[49] MarkLogic Corporation. MarkLogic Server. `http://www.marklogic.com/`.

[50] Microsoft Corporation. Microsoft Silverlight. `http://www.microsoft.com/silverlight/`.

[51] Microsoft Corporation. Windows Presentation Foundation (WPF). `http://msdn.microsoft.com/en-us/library/ms754130.aspx`.

[52] J. Miller and J. Mukerji. *MDA Guide Version 1.0.1*. Object Management Group, 2003.

[53] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML Schema Languages using Formal Language Theory. *ACM Trans.*, 5(4):660–704, 2005.

[54] M. Nečaský. *Conceptual Modeling for XML*, volume 99 of *Dissertations in Database and Information Systems*. IOS Press, Amsterdam, Netherlands, 2009.

[55] M. Nečaský. *Conceptual Modeling for XML*, volume 99 of *Dissertations in Database and Information Systems Series*. IOS Press/AKA Verlag, January 2009.

[56] M. Nečaský, J. Klímek, J. Malý, and I. Mlýnková. Evolution and Change Management of XML-based Systems. *Journal of Systems and Software*, 85(3), 2012.

[57] 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.

[58] M. Nečaský and I. Mlýnková. On Different Perspectives of XML Schema Evolution. In *FlexDBIST'09: Proc. of the 5th Int. Workshop on Flexible Database and Information System Technology*, Linz, Austria, 2009. IEEE Computer Society.

[59] M. Nečaský and I. Mlýnková. Five-Level Multi-Application Schema Evolution. In *DATESO '09: Proc. of the Databases, Texts, Specifications, and Objects*, pages 213–217. MatfyzPress, April 2009.

[60] M. Nečaský and I. Mlýnková. A Framework for Efficient Design, Maintaining, and Evolution of a System of XML Applications. In *DATESO '10: Proc. of the Databases, Texts, Specifications, and Objects*, pages 38 – 49. MatfyzPress, April 2010.

[61] M. Nečaský, I. Mlýnková, J. Klímek, and J. Malý. When conceptual model meets grammar: A dual approach to XML data modeling. *Data & Knowledge Engineering*, 72:1 – 30, 2012.

[62] M. Nečaský and I. Mlýnková. Five-Level Multi-Application Schema Evolution. In *DATESO '09*, pages 90–104, 2009.

[63] M. Necasky. Reversee Engineering of XML Schemas to Conceptual Diagrams. In *Proceedings of The Sixth Asia-Pacific Conference on Conceptual Modelling*, pages 117–128, Wellington, New Zealand, January 2009. Australian Computer Society.

[64] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Trans. Internet Technol.*, 2(2):151–185, May 2002.

[65] National information exchange model. `http://www.niem.gov/`.

[66] Object Management Group. Object Constraint Language Specification 2.3.1, 2012. `http://www.omg.org/spec/OCL/2.3.1/`.

[67] Oracle Corporation. Oracle XML DB Developer's Guide – XML Schema Evolution. `http://download-uk.oracle.com/docs/cd/B28359_01/appdev.111/b28369/xdb07evo.htm#BCGFEEBB`.

[68] Oracle Corporation. Oracle XML DB Home. `http://www.oracle.com/technetwork/database/features/xmldb/index.html`.

[69] T. Parr. ANTLR – ANother Tool for Language Recognition. `http://antlr.org`.

[70] M. Polák, I. Mlýnková, and E. Pardede. XML Query Adaptation as Schema Evolves. *ISD 2013*, 2013.

[71] E. Qeli, J. Gllavata, and B. Freisleben. Customizable Detection of Changes for XML Documents using XPath Expressions. In D. C. A. Bulterman and D. F. Brailsford, editors, *Proceedings of the 2006 ACM Symposium on Document Engineering*, pages 88–90, Amsterdam, Netherlands, October 2006. ACM Press.

[72] M. Raghavachari and O. Shmueli. Efficient Revalidation of XML Documents. *IEEE Trans. on Knowl. and Data Eng.*, 19:554–567, April 2007.

[73] E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal*, 10(4):334–350, 2001.

[74] M. Richters and M. Gogolla. OCL: Syntax, Semantics, and Tools. In *Object Modeling with the OCL*, volume 2263 of *LNCS*, pages 447–450. Springer-Verlag, 2002.

[75] N. Routledge, L. Bird, and A. Goodchild. UML and XML Schema. In *Proceedings of 13th Australasian Database Conference (ADC 2002)*. ACS, 2002.

[76] Saxonica. Saxon xslt processor 9.4, 2012. `http://saxon.sourceforge.net/`.

[77] H. Su, D. K. Kramer, and E. A. Rundensteiner. XEM: XML Evolution Management. Technical Report WPI-CS-TR-02-09, Worcester Polytechnnic Institut, Computer Science Department, Worcester Polytechnnic Institute, Worcester, Massachusetts, 2002.

[78] M. Tan and A. Goh. Keeping Pace with Evolving XML-Based Specifications. In *EDBT'04 Workshops*, pages 280–288, Berlin, Heidelberg, 2005. Springer.

[79] Technische Universität Dresden. Dresden OCL 3.2.0. `http://www.dresden-ocl.org`.

[80] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures (Second Edition)*. W3C, 2004.

[81] W3C. Document object model (dom). `http://www.w3.org/DOM/`.

[82] W3C. Document Type Declaration, 2000. `http://www.w3.org/TR/xml11/`.

[83] W3C. XML Linking Language (XLink) Version 1.0, 2001. `http://www.w3.org/TR/xlink/`.

[84] W3C. RDF/XML Syntax Specification (Revised), 2004. `http://www.w3.org/TR/rdf-syntax-grammar/`.

[85] W3C. Web Services Architecture, 2004. `http://www.w3.org/TR/ws-arch/`.

[86] W3C. Web services description language (wsdl) version 2.0, 2007. `http://www.w3.org/TR/wsdl20/`.

[87] W3C. XSL Transformations (XSLT) Version 2.0, 2007. `http://www.w3.org/TR/xslt20/`.

[88] W3C. SPARQL Query Language for RDF, 2008. `http://www.w3.org/TR/rdf-sparql-query/`.

[89] W3C. XML Path Language (XPath) 2.0, 2010. `http://www.w3.org/TR/xpath20/`.

[90] W3C. XProc: An XML Pipeline Language, 2010. http://www.w3.org/TR/xproc/.

[91] W3C. XQuery Update Facility 1.0, 2011. http://www.w3.org/TR/xquery-update-10/.

[92] W3C. XML Schema 1.1, 2012. http://www.w3.org/TR/xmlschema-1/.

[93] W3C. XSL Transformations (XSLT) Version 3.0, Working Draft 10, 2012. http://www.w3.org/TR/xslt-30/.

[94] W3C. Xml path language (xpath) 3.0, candidate recommendation, 2013. http://www.w3.org/TR/xpath-30/.

[95] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-diff: An effective change detection algorithm for xml documents. *Data Engineering, International Conference on*, 0:519, 2003.

[96] F. Wenfei and S. Jerome. Integrity Constraints for XML. *Journal of Computer and System Sciences (JCSS)*, 66(1):254–291, feb 2003.

[97] A. Wojnar, I. Mlýnková, and J. Dokulil. Structural and Semantic Aspects of Similarity of Document Type Definitions and XML Schemas. *Information Sciences*, 180(10):1817–1836, 2010. Special Issue on Intelligent Distributed Information Systems.

Appendices

A. XSDs for the Sample Schemas

Listing A.1: XSD for the PSM schema from Fig. 3.1 – version v

```
<?xml version='1.0'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
  version='1.1' elementFormDefault='qualified'>

  <xs:element name='purchase' type='Purchase' />

  <xs:complexType name='Purchase'>
    <xs:sequence>
      <xs:element name='customer' type='Customer'/>
      <xs:element name='items' type='Items'/>
      <xs:element name='delivery' type='Address'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Customer'>
    <xs:sequence>
      <xs:element name='name'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Items'>
    <xs:sequence>
      <xs:element name='Item' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Item'>
    <xs:sequence>
      <xs:element name='code'/>
      <xs:element name='price'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Address'>
    <xs:sequence>
      <xs:element name='state'/>
      <xs:element name='street'/>
      <xs:element name='city'/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing A.2: XSD for the PSM schema from Fig. 3.1 – version $\tilde{v}$

```
<?xml version='1.0'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
  version='1.1' elementFormDefault='qualified'>

  <xs:element name='purchase' type='Purchase' />
  <xs:complexType name='Purchase'>
    <xs:sequence>
      <xs:element name='purchase-date' type='xs:date'/>
      <xs:element name='customer-info' type='CustomerInfo'/>
      <xs:element name='item' type='Item' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='CustomerInfo'>
    <xs:sequence>
      <xs:element name='customer' type='Customer'/>
      <xs:element name='address' type='Address' minOccurs='0'/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name='Customer'>
    <xs:sequence>
      <xs:element name='customer-no' type='xs:integer'/>
      <xs:element name='email' type='xs:string'
        minOccurs='0' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Item'>
    <xs:sequence>
      <xs:element name='product' type='Product'/>
      <xs:element name='qty' type='ItemI'/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name='ItemI'>
    <xs:sequence>
      <xs:element name='amount' type='xs:integer'/>
      <xs:element name='unit-price' type='xs:integer'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Address'>
    <xs:sequence>
      <xs:element name='zip'/>
      <xs:element name='city'/>
      <xs:element name='street'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Product'>
    <xs:sequence>
      <xs:element name='code' type='xs:integer'/>
      <xs:element name='subcode' type='xs:integer'/>
      <xs:element name='title' type='xs:string'/>
      <xs:element name='weight' type='xs:integer'/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing A.3: XSD for the PSM schema from Fig. 4.3 – version v

```
<?xml version='1.0'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
  version='1.1' elementFormDefault='qualified'>

  <xs:element name='purchase' type='Purchase' />

  <xs:complexType name='Purchase'>
    <xs:sequence>
      <xs:element name='customer' type='Customer'/>
      <xs:element name='items' type='Items'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Customer'>
    <xs:sequence>
      <xs:element name='name'/>
      <xs:element name='delivery' type='Address'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Items'>
    <xs:sequence>
      <xs:element name='Item' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Item'>
    <xs:sequence>
      <xs:element name='code'/>
      <xs:element name='price'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Address'>
    <xs:sequence>
      <xs:element name='state'/>
      <xs:element name='street'/>
      <xs:element name='city'/>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

Listing A.4: XSD for the PSM schema from Fig. 4.3 – version $\tilde{v}$

```
<?xml version='1.0'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
  version='1.1' elementFormDefault='qualified'>

  <xs:element name='purchase' type='Purchase'/>

  <xs:complexType name='Purchase'>
    <xs:sequence>
      <xs:element name='purchase-date' type='xs:date'/>
      <xs:element name='customer-info' type='CustomerInfo'/>
      <xs:element name='item' type='Items' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='CustomerInfo'>
    <xs:sequence>
      <xs:element name='customer' type='Customer'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Customer'>
    <xs:sequence>
      <xs:element name='customer-no' type='xs:integer'/>
      <xs:element name='delivery-address' type='Address' minOccurs='0'/>
      <xs:element name='emails' type='CustEmail'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Items'>
    <xs:sequence>
      <xs:element name='item' type='Item' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Item'>
    <xs:sequence>
      <xs:element name='product' type='Product'/>
      <xs:element name='qty' type='ItemI'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Address'>
    <xs:sequence>
      <xs:element name='zip'/>
      <xs:element name='city'/>
      <xs:element name='street'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Product'>
    <xs:sequence>
      <xs:element name='code' type='xs:integer'/>
      <xs:element name='subcode' type='xs:integer'/>
      <xs:element name='title' type='xs:string'/>
      <xs:element name='weight' type='xs:integer'/>
    </xs:sequence>
  </xs:complexType>
```

```
  <xs:complexType name='ItemI'>
    <xs:sequence>
      <xs:element name='amount' type='xs:integer'/>
      <xs:element name='unit-price' type='xs:integer'/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name='CustEmail'>
    <xs:sequence>
      <xs:element name='email' type='xs:string'
        minOccurs='0' maxOccurs='5'/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing A.5: XSD for the PSM schema from Fig. 5.2

```
<?xml version='1.0'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
  version='1.1' elementFormDefault='qualified'>
  <xs:element name='org' type='Organization'/>
  <xs:complexType name='Organization'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
      <xs:element name='budget' type='xs:double'/>
      <xs:element name='dpt' type='Department' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name='Department'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
      <xs:element name='budget' type='xs:double'/>
      <xs:element name='tm' type='Team' minOccurs='0' maxOccurs='unbounded'/>
      <xs:element name='emp' type='Employee' maxOccurs='unbounded'/>
      <xs:element name='int' type='Intern' minOccurs='0' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name='Team'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
      <xs:element name='mem' type='Member' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name='Member'>
    <xs:attribute name='empNo' type='xs:string'/>
  </xs:complexType>
  <xs:complexType name='Employee'>
    <xs:sequence>
      <xs:element name='firstName' type='xs:string'/>
      <xs:element name='lastName' type='xs:string'/>
      <xs:element name='salary' type='xs:double'/>
    </xs:sequence>
    <xs:attribute name='empNo' type='xs:string'/>
  </xs:complexType>
  <xs:complexType name='Intern'>
    <xs:sequence>
      <xs:element name='firstName' type='xs:string'/>
      <xs:element name='lastName' type='xs:string'/>
    </xs:sequence>
    <xs:attribute name='empNo' type='xs:string'/>
  </xs:complexType>
</xs:schema>
```

Listing A.6: XSD for the PSM schema from Fig. 5.3a

```
<?xml version='1.0'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
  version='1.1' elementFormDefault='qualified'>
  <xs:element name='organizations' type='Organization'/>
  <xs:complexType name='Organization'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
      <xs:element name='budget' type='xs:double'/>
      <xs:element name='project' type='Project' minOccurs='0' maxOccurs='unbounded'/>
      <xs:element name='dpt' type='Department' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Project'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
      <xs:element name='budget' type='xs:double'/>
      <xs:element name='team' type='Team' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Department'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
      <xs:element name='budget' type='xs:double'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Team'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
      <xs:element name='host' type='Host'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Host'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing A.7: XSD for the PSM schema from Fig. 5.3b

```
<?xml version='1.0'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
  version='1.1' elementFormDefault='qualified'>
  <xs:element name='teams' type='Team'/>
  <xs:complexType name='Team'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
      <xs:element name='host' type='Department'/>
      <xs:element name='member' type='Employee' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Department'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
      <xs:element name='budget' type='xs:double'/>
      <xs:element name='organization' type='Organization'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Organization'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
      <xs:element name='budget' type='xs:double'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Employee'>
    <xs:sequence>
      <xs:element name='empNo' type='xs:string'/>
      <xs:element name='salary' type='xs:double'/>
      <xs:element name='employer' type='Employer'/>
      <xs:element name='internship' type='Internship' minOccurs='0'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Employer'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Internship'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing A.8: XSD for the PSM schema from Fig. 5.8

```
<?xml version='1.0'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
  version='1.1' elementFormDefault='qualified'>
  <xs:element name='organization' type='Organization'/>

  <xs:complexType name='Organization'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
      <xs:element name='departments' type='Departments'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Departments'>
    <xs:sequence>
      <xs:element name='department' type='Department'
        minOccurs='0' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Department'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
      <xs:element name='employees' type='Employees' minOccurs='0'/>
      <xs:element name='manager' type='Manager' minOccurs='0'/>
      <xs:element name='subdepartments' type='Subdepartments'
        minOccurs='0'/>
      <xs:element name='interns' type='Interns' minOccurs='0'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Employees'>
    <xs:sequence>
      <xs:element name='employee' type='Employee' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Manager'>
    <xs:complexContent>
      <xs:extension base='Employee'/>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name='Subdepartments'>
    <xs:sequence maxOccurs='unbounded'>
      <xs:element name='department' type='Department'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Employee'>
    <xs:sequence>
      <xs:element name='empNo' type='xs:string'/>
      <xs:element name='firstName' type='xs:string'/>
      <xs:element name='lastName' type='xs:string'/>
      <xs:element name='salary' type='xs:double'/>
      <xs:element name='phone' type='xs:string'
        minOccurs='0' maxOccurs='1'/>
    </xs:sequence>
```

```
  </xs:complexType>

  <xs:complexType name='Interns'>
    <xs:sequence>
      <xs:element name='Intern' type='EmployeeI'
        maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='EmployeeI'>
    <xs:sequence>
      <xs:element name='empNo' type='xs:string'
        minOccurs='1'/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing A.9: XSD for the PSM schema from Fig. 6.1 – version v

```
<?xml version='1.0'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
  version='1.1' elementFormDefault='qualified'>

  <xs:element name='customer' type='Customer'/>

  <xs:complexType name='Customer'>
    <xs:sequence>
      <xs:element name='name' type='xs:string'/>
      <xs:element name='purchase' type='Purchase' maxOccurs='unbounded' />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Purchase'>
    <xs:sequence>
      <xs:element name='status' type='xs:string'/>
      <xs:element name='code' type='xs:string'/>
      <xs:element name='item' type='Item' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Item'>
    <xs:sequence>
      <xs:element name='code' type='xs:string'/>
      <xs:element name='price' type='xs:double'/>
      <xs:element name='quantity' type='xs:integer'/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing A.10: XSD for the PSM schema from Fig. 6.1 – version $\tilde{v}$

```
<?xml version='1.0'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
  version='1.1' elementFormDefault='qualified'>

  <xs:element name='customer' type='Customer'/>

  <xs:complexType name='Customer'>
    <xs:sequence>
      <xs:element name='firstName' type='xs:string'/>
      <xs:element name='lastName' type='xs:string'/>
      <xs:element name='realizedPurchases' type='RealizedList'/>
      <xs:element name='pendingPurchases' type='PendingList'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='RealizedList'>
    <xs:sequence>
      <xs:element name='purchase' type='Purchase' minOccurs='0' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='PendingList'>
    <xs:sequence>
      <xs:element name='purchase' type='Purchase' minOccurs='0' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='Purchase'>
    <xs:sequence>
      <xs:element name='totalPrice' type='xs:double'/>
      <xs:element name='item' type='Item' maxOccurs='unbounded'/>
    </xs:sequence>
    <xs:attribute name='code' type='xs:string'/>
  </xs:complexType>

  <xs:complexType name='Item'>
    <xs:sequence>
      <xs:element name='price' type='xs:double'/>
      <xs:element name='quantity' type='xs:integer'/>
    </xs:sequence>
    <xs:attribute name='code' type='xs:string'/>
  </xs:complexType>
</xs:schema>
```

B. Adaptation Scripts for the Sample Scenarios

Listing B.1: Adaptation sript for Fig. 4.3

```
<xsl:template match='/purchase'>
  <purchase>
    <xsl:apply-templates select='purchase-date'/>
    <xsl:apply-templates select='customer-info'/>
    <xsl:call-template name='purchase-items'/>
  </purchase>
</xsl:template>

<xsl:template match='/purchase/customer-info'>
  <customer-info>
    <xsl:apply-templates select='customer'/>
  </customer-info>
</xsl:template>

<xsl:template match='/purchase/customer-info/customer'>
  <customer>
    <xsl:apply-templates select='customer-no'/>
    <xsl:apply-templates select='../address'/>
    <xsl:call-template name='emails'/>
  </customer>
</xsl:template>

<xsl:template match='/purchase/customer-info/address'>
  <delivery-address>
    <xsl:apply-templates select='city'/>
    <xsl:apply-templates select='street'/>
    <xsl:apply-templates select='zip'/>
  </delivery-address>
</xsl:template>

<xsl:template name='emails'>
  <emails>
    <xsl:copy-of select='email[position() &lt;= 5]'/>
  </emails>
</xsl:template>

<xsl:template name='purchase-items'>
  <items>
    <xsl:apply-templates select='item'/>
  </items>
</xsl:template>

<xsl:template match='/purchase/item/product'>
  <product>
    <xsl:apply-templates select='code|subcode|title'/>
  </product>
</xsl:template>
```

```
<!-- blue nodes template -->
<xsl:template match='/purchase/item'>
  <xsl:copy>
    <xsl:copy-of select='@*'/>
    <xsl:apply-templates select='*'/>
  </xsl:copy>
</xsl:template>

<!-- green nodes template -->
<xsl:template match='/purchase/purchase-date
  | /purchase/customer-info/customer/*
  | /purchase/item/product/*[.= ../code|../subcode|../title]
  | /purchase/item/qty/*
  | /purchase/customer-info/address/*'>
    <xsl:copy-of select='.'/>
</xsl:template>
```

Listing B.2: Adaptation sript for Fig. 6.1

```
<xsl:stylesheet version='3.0' xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
 <xsl:template match='/customer'>
  <customer>
    <xsl:variable name='firstName-new' select= 'tokenize(name,'\s+')[1]' />
    <xsl:variable name='lastName-new' select= 'tokenize(name,'\s+')[2]' />
    <firstName>
     <xsl:sequence select='$firstName-new'/>
    </firstName>
    <lastName>
     <xsl:sequence select='$lastName-new'/>
    </lastName>
    <xsl:call-template name='RealizedList'/>
    <xsl:call-template name='PendingList'/>
  </customer>
 </xsl:template>

 <xsl:template name='RealizedList'>
  <realizedPurchases>
   <xsl:variable name='purchase-new'
     select="for $p in purchase[status eq 'realized'] return oclX:apply-templates($p)"/>
   <xsl:sequence select="$purchase-new"/>
  </realizedPurchases>
 </xsl:template>

 <xsl:template name='PendingList'>
  <pendingPurchases>
   <xsl:variable name='purchase-new'
     select="for $p in purchase[status eq 'pending'] return oclX:apply-templates($p)"/>
   <xsl:sequence select="$purchase-new"/>
  </pendingPurchases>
 </xsl:template>

 <xsl:template match='/customer/purchase'>
  <purchase>
   <xsl:apply-templates select='code' />
   <totalPrice/>
   <xsl:apply-templates select='item' />
  </purchase>
 </xsl:template>

 <xsl:template match='/customer/purchase/item'>
  <item>
   <xsl:apply-templates select='code'/>
   <xsl:apply-templates select='price'/>
   <xsl:apply-templates select='quantity'/>
  </item>
 </xsl:template>

  <xsl:template priority='0' match='item/code | purchase/code' >
   <xsl:attribute name='{name()}'>
    <xsl:value-of select='.' />
   </xsl:attribute>
  </xsl:template>
  <xsl:template priority='0' match='price | quantity' >
   <xsl:copy-of select='.' />
  </xsl:template>
 </xsl:stylesheet>
```

C. Schematron Schemas for the Sample Constraints

Listing C.1: Translation of constraints for OrganizationSchema from Fig. 5.2

```
<!-- Constraints for OrganizationSchema from Figure 5.2 -->
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron"/>
 <sch:pattern id="Organization">
  <sch:rule context="/org">
   <sch:let name="self" value="." />
   <!-- PSM IC1 -->
   <sch:assert test="oclX:forAll($self/dpt/tm, function($t)
                                { count(t/mem) < 0.1 * count($self/dpt/emp) } )" />
  </sch:rule>
  <sch:rule context="/org">
   <sch:let name="o" value="." />
   <!-- PSM IC7 -->
   <sch:assert test="
      let $internships := oclX:iterate($o/dpt/emp, (), function($e, $acc) {
                oclX:including($acc, map {
                  'employee' = $e,
                  'departments' = oclX:select($o/dpt, function($d) {
                        oclX:includes($d/int/@empNo, $e/@empNo) } ) } ) } )
      return oclX:forAll($internships, function($i) { count($i('departments')) lt 3 })" />
  </sch:rule>
 </sch:pattern>

 <sch:pattern id="Intern">
  <sch:rule context="/org/dpt/int">
   <sch:let name="i" value="." />
   <!-- PSM IC3 -->
   <sch:assert test="not( $i/.. is
           (for $p in $i return //emp[./@empNo eq $p/@empNo]/..) )" />
  </sch:rule>
 </sch:pattern>
</sch:schema>
```

Listing C.2: Translation of constraints for OrganizationProjectsSchema from Fig. 5.3a

```
<!-- Constraints for OrganizationProjectsSchema from Figure 5.3a -->
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron"/>
 <sch:pattern id="Organization">
  <sch:rule context="/organization">
   <sch:let name="self" value="." />
   <!-- PSM IC2 -->
   <sch:assert test="$self/budget le sum($self/project/budget)" />
  </sch:rule>
 </sch:pattern>

 <sch:pattern id="Project">
  <sch:rule context="/organization">
   <sch:let name="self" value="." />
   <!-- PSM IC4 -->
   <sch:assert test="oclX:forAll($self/team, function($t) {
       (for $p in $t/host return //department[./name eq $p/@name])/.. is $self/.. } )" />
  </sch:rule>
 </sch:pattern>

</sch:schema>
```

Listing C.3: Translation of constraints for TeamsSchema from Fig. 5.3b

```
<!-- Constraints for TeamsSchema from Figure 5.3b -->
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron"/>
 <sch:pattern id="Teams">
  <sch:rule context="/teams">
   <sch:let name="self" value="." />
   <!-- PSM IC6 -->
   <sch:assert test="oclX:forAll($self/team/member,
       function($m) { count(oclX:select($self/team,
           function($t) { oclX:includes($t/member/empNo, $m/empNo) } )) lt 5})" />
  </sch:rule>
 </sch:pattern>
</sch:schema>
```

Listing C.4: Translation of constraints from the organization hierarchy schema (Figure 5.8

```
<!-- Constraints for OrganizationHierarchySchema from Figure 5.8 -->
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron">
 <sch:pattern id="Department">
  <sch:rule context="/organization/departments/descendant::department">
   <sch:let name="d" value="." />
   <sch:assert test="let $count := count(oclX:collect(oclX:collect(
         oclX:closure(.,function($sd) { $sd/subdepartments/department }),
         function($d) { $d/employees }), function($e) { $e/employee }))
       return if (count(interns/intern) gt 0) then $count ge 3 else true()">
       Only departments with at least 3 employees can accept interns,
       department <sch:value-of select="$d/name" /> has less employee(s)
   </sch:assert>
  </sch:rule>
 </sch:pattern>

 <sch:pattern id="Employee" abstract="true">
  <sch:rule context="$e">
   <sch:assert test="empNo ne '' "/> <!-- empNo is not an empty string -->
  </sch:rule>
 </sch:pattern>

 <sch:pattern id="EmployeeI">
  <sch:rule context="//intern">
   <sch:let name="e" value="." />
   <sch:assert test="if (exists(../..)) then
         not(../.. is (for $e2 in . return
           (//manager | //employee)[./empNo = $e2/empNo])/parent::employees/..)
       else true()">
       Internship in home department is forbidden
   </sch:assert>
  </sch:rule>
 </sch:pattern>

 <sch:pattern id="Manager">
  <sch:rule context="//manager">
   <sch:let name="m" value="." />
   <sch:assert test="oclX:includes(oclX:collect(../employees/employee,
       function($e) { data($e/empNo) }), data(empNo))">
     Manager is an employee of its department
   </sch:assert>
   <sch:assert test="exists(phone)">
     Managers must state their phone numbers
   </sch:assert>
  </sch:rule>
 </sch:pattern>

 <!--instance pattern for PSMClass: Manager's ancestor Employee-->
 <sch:pattern id="Manager-as-Employee" is-a="Employee">
  <sch:param name="e" value="//manager" />
 </sch:pattern>
 <!--instance pattern for PSMClass: "Employee"-->
 <sch:pattern id="Employee-as-Employee" is-a="Employee">
  <sch:param name="e" value="//employee" />
 </sch:pattern>
</sch:schema>
```

Listing C.5: Translation of selected constraints from the organization hierarchy schema (Figure 5.8) with applied rewritings

```
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron">
 <sch:pattern id="Department">
  <sch:rule context="/organization/departments/descendant::department">
   <sch:let name="d" value="." />
   <sch:assert test="let $count :=
         count(./descendant-or-self::department/employees/employee)
       return if (count(interns/intern) gt 0) then $count ge 3 else true()">
       Only departments with at least 3 employees can accept interns,
       department <sch:value-of select="$d/name" /> has less employee(s)
   </sch:assert>
  </sch:rule>
 </sch:pattern>

 <sch:pattern id="Manager">
  <sch:rule context="//manager">
   <sch:let name="m" value="." />
   <sch:assert test="../employees/employee/empNo = $m/empNo">
     Manager is an employee of its department
   </sch:assert>
  </sch:rule>
 </sch:pattern>
</sch:schema>
```