Charles University in Prague
Faculty of Mathematics and Physics

DIPLOMA THESIS

Bc. Lukáš Berka

Profiling Translation of Conceptual Schemas to XML Schemas

Department of Software Engineering

Supervisor: Mgr. Martin Nečaský, Ph.D.

Study Program: Computer Science, Software Systems

2010

In this part, I would like to express my gratitude to all who supported my efforts during months of my work on this diploma thesis.

Thanks go to my supervisor, Mgr. Martin Nečaský, Ph.D., for his kindness as well as valuable and inspiring suggestions.

I would like to thank my family and my girlfriend Alena for their devoted support during my studies.

Title: Profiling Translation of Conceptual Schemas to XML Schemas
Author: Bc. Lukáš Berka
Department: Department of Software Engineering
Supervisor: Mgr. Martin Nečaský, Ph.D.
Supervisor's e-mail address: necasky@ksi.mff.cuni.cz

Abstract: In the present work we analyze the algorithm that was introduced in [4]. The algorithm performs a translation of a conceptual schema to an XML schema expressed in the XML Schema language. We look for limitations of the algorithm and try to discover parameters that can be potentially used to influence its behavior. We propose solutions to the most serious limitations. Also, we introduce a concept of a translation profiling. The concept is based on a configuration that contains a set of parameters. We modify the algorithm to use the user requirements specified in the configuration.

Thanks to the improvements, the new algorithm works with the concept of XML Namespaces, uses XML Schema designs and also, focuses on an elimination of redundancy. The elimination of redundancy in an output of the algorithm is an important part of this work and we create a formal model that helps us to solve this task.

Keywords: XML, XML Schema, conceptual modeling, profiling

Název práce: Profilování překladu konceptuálních schémat do XML schémat
Autor: Bc. Lukáš Berka
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: Mgr. Martin Nečaský, Ph.D.
e-mail vedoucího: necasky@ksi.mff.cuni.cz

Abstrakt: Práce se zabývá analýzou algoritmu pro převod konceptuálních schémat do XML schémat v jazyce XML Schema, který byl uveden v [4]. Snažíme se najít nedostatky tohoto algoritmu a také hledáme parametry, kterými by bylo možné jeho běh ovlivnit. Na základě zjištěných poznatků poté navrhujeme jeho vylepšení. Uvádíme také tzv. koncept profilování překladu. Tento koncept je založen na množině parametrů, zvané konfigurace. Původní algoritmus je podle něj upraven tak, aby zohledňoval požadavky, které uživatel zadá skrze svou konfiguraci.

Díky všem našim úpravám je algoritmus schopný pracovat se jmennými prostory XML, s návrhovými vzory pro jazyk XML Schema a ve svém výstupu omezuje výskyt určitých typů redundancí. Úkol omezit redundance ve výstupu algoritmu je důležitou součástí této práce a proto vytváříme formální model, který nám s řešením tohoto problému velmi pomáhá.

Klíčová slova: XML, XML Schema, konceptuální modelování, profilování

Contents

Chapter 1

Introduction

Because of its simplicity, versatility and platform independence, XML [1] has become a popular format for exchanging and representing structured and semi-structured data [15][16]. As the amount of data represented in XML grows, designing its representations effectively is becoming more and more important.

The representations of XML data are usually given by XML schemas and to create them, we apply XML schema languages such as XML Schema [2], RELAX NG [3], Schematron [17] or DTD [1]. Each XML schema language has its own specifics, advantages and disadvantages [18]. XML schemas represent logical models of XML documents.

When designing a new XML data representation its not easy to work on the logical level only. The languages mentioned above provide a lot of constructs for describing the structure of XML documents, but we can hardly express semantics of data in them. A maintenance of created logical models brings the same problem.

Constructing an XML schema in an XML schema language is difficult process, because such languages are extensive and ambiguous, and non-technical users cannot participate in it. Moreover, non-technical user can hardly understand XML schemas written in that languages.

Therefore, we need to work on a higher level of abstraction first.

The described situation can be liken to the one in the world of relational databases. Its not easy to design model of relational data directly on a logical level, i.e. to design schemas of relational tables directly. Therefore, in a process of designing relational databases we work on a conceptual level first.

There is the Entity-Relationship (E-R) [5] model for the conceptual modeling of relational data. Once the E-R model is constructed we can start creating schemas of concrete tables.

The conceptual modeling is useful for designing of XML data representation as well. The E-R model cannot be used for that purpose, because XML as a logical database model has some special differences, e.g. XML has hierarchical and irregular structure, ordering on siblings and mixed content. The E-R model cannot deal with these special features.

Therefore, some extensions of the E-R model has been developed. However, none of the extensions can express hierarchical structure of XML data. On the other hand, models designed to express the hierarchical structure often have problems with another XML features. A brief description of some models as well as their comparison can be found in [4].

In present work, we are interested in a conceptual model for XML data modeling called XSEM [4]. It consists of two parts, XSEM-ER and XSEM-H models. The first part is an extension of the E-R model known from relational databases, the second is used to express the hierarchical principle of XML data.

The conceptual model XSEM is implemented in some tools for XML data modeling, e.g. in XCase tool [10].

After we create an XML schema on a conceptual level, it is good to design data representation at a lower level, the logical one. A translation of an XML schema from the conceptual level to the logical one can be done automatically. An example of algorithm realizing such a translation can be found in [4].

However, the algorithm introduced in [4] is just a basic algorithm that needs improvements. Another problem of the algorithm is that its process is fully automatical and cannot be influenced by preset parameters.

1.1 Contribution

In this thesis, we introduce a concept of profiling a translation from a conceptual model XSEM to an XML schema expressed in the XML Schema language.

Thanks to the concept, everyone who wants to use the algorithm, described in [4], is able to get the result that best suits his needs. The only thing the user must do is to create his own configuration for the translation process. The configuration can be also called the *user profile*.

We assume that once the user creates his profile, he should be able to save it in a persistent storage and then, load it and reuse it at any time.

Another benefit of this work are improvements to the original translation algorithm. Some of them are simple, some are complex, but all the improvements are clearly described.

For example, thanks to the improvements the new version of the algorithm works with the concept of XML Namespaces [14], uses XML Schema designs [13] and also, focuses on an elimination of redundancy.

The description of the solution of the redundancy elimination problem is considered to be an important part of this work. There are several types of redundancy in an arbitrary output of the translation algorithm. For some of them, we create a formal model that helps us to design algorithms for redundancy elimination.

1.2 Outline

The rest of this work is organized as follows. In Chapter 2, we focus on a conceptual modeling for XML, on its specifics and typical problems. We describe the conceptual model XSEM in brief, we discuss which items it typically contains.

In Chapter 3, we describe a basic algorithm for translation of XML schemas from the conceptual level into the XML Schema language. The algorithm was originaly introduced in [4]. In this work, we describe it only in brief. In detail, we focus on its limitations and show an XML schema, against which all of its outputs are valid.

Chapter 4 is intended to bring some improvements proposals of the algorithm described in Chapter 3. Moreover, in Chapter 4, we propose a concept of profiling the translation process that helps users get results that best suits their needs.

In Chapter 5, we introduce the most important improvement of the basic translation algorithm. We propose the algorithms for redundancy elimination there.

Finally, Chapter 6 concludes and provides some future directions in our research.

Chapter 2

Conceptual Modeling for XML

When designing an XML data representation we usually start on a pretty high level of abstraction called a conceptual level. On that level we try to create a conceptual model that describes the XML data independently of its representation in XML documents.

That process is analogical to the one in the world of relational databases, where we have a relational model as the logical one and the Entity-Relationship (E-R) model [5] as the conceptual one.

There are several approaches to a conceptual modeling for XML. Conceptual models can be based on the E-R model, the model of UML class diagrams, an Object Role Modeling (ORM) or we can use a hierarchical approach for XML data modeling.

Both the first two groups consist of models that are based on extending the original models with new modeling constructs. With those constructs, the models can express special features of XML, such as hierarchical and irregular structure of a modeled data, mixed content or ordering on siblings.

The models based on the E-R model are especially these: Extended E-R [6], EReX [7], EER [8], XER [9], etc.

Using this approach, the XML representation of the data is modeled directly on the conceptual level. That means, when designing the conceptual schema we have to concentrate on XML specific implementation details which is not easy especially for non-technical users.

The UML-based approaches to conceptual modeling of XML usually apply Model-Driven Architecture (MDA) [11]. According to MDA, a conceptual schema should be modeled in Platform-Independent Model (PIM) first. PIM describes data independently of representations in XML documents.

After that, an XML schema in a Platform-Specific Model (PSM) should be designed. The PSM specifies how the components of the PIM model are represented in a given type of XML documents.

Several PSM schemas can be derived from the same PIM schema. Such a derivation is fully automatical and that is the problem. PSM schemas should be designed with respect to user requirements which cannot be done automaticaly.

A brief description of some conceptual models for XML as well as their comparison can be found in [4]. In present work, we are interested in the conceptual model for XML data modeling called XSEM, which was introduced in [4].

In Section 2.1, we describe the XSEM model in brief. In Section 2.2, we present a tool for XML data modeling called XCase, which implements the mentioned conceptual model.

2.1 Conceptual Model XSEM

The XSEM model is based on the idea that the XML schema design process must be composed of two parts to be applicable in practice.

The first part should care about modeling the data on the conceptual level abstracted from its representation in XML. The second part should model how the data is represented in different types of XML documents.

In XSEM, the strict separation of the modeling process is achieved by dividing the model to two parts: XSEM-ER and XSEM-H. We focus on these parts in Section 2.1.1 and Section 2.1.2.

2.1.1 XSEM-ER

XSEM-ER is a conceptual model, extending the E-R model [5], that is not influenced by XML schema languages or special features of an XML data model.

It is clear and self-describing, which makes non-technical users (investors, stakeholders) able to understand it and participate on the process of XML schema design.

The XSEM-ER is a platform-independent model (PIM) from the Model-Driven Architecture (MDA) [11] point of view.

As an extension of E-R model, XSEM-ER provides two basic modeling constructs: entity type and relationship type.

The **entity types** model sets of real-world objects - entities. The entities with the same semantics (e.g. all customers) are modeled by the same entity type. Each entity type has a name and some attributes, that model properties of corresponding entities. There can not be two attributes with the same name in the same entity type.

Also, we have so-called weak entity types. There is only one difference between the ordinary (strong) entity types and the weak ones. The weak entity types can't be stand-alone, they always depend on existence of another entity type. The definition of weak entity type includes a list of all entity types, which the weak type depends on.

In E-R model each entity type (both strong and weak) has a key, which is a subset of its attributes. Attributes of an entity type can be simple, composite, multivalued or choice. In contrast, in XSEM-ER model only simple attributes are considered and entity type keys are not considered at all.

The **relationship types** model relationships between entities, e.g. the fact that a customer makes a purchase. Like entity types, relationship types have their names and attributes. Relationships with the same semantics are grouped to a relationship set, which is modeled as a relationship type in E-R model.

Two or more entities may participate in the relationship. Each of them may have its role assigned. Especially, the roles are useful when a given entity type participates in the same relationship type twice or more times. In that case, such participations are distinguished by the roles.

The relationship types with more than two participants are allowed. It is necessary to constraint the number of instances of a relationship type in which an instance of a given entity type can appear. For this, cardinality constraints are applied. A cardinality constraint is a pair (m, n), where m and n are natural numbers including 0 and $*$ (infinity) and m is lower than n.

The E-R model has an extension called **IS-A hierarchies**, which is used to classify instances of a general entity type G to more specific entity types

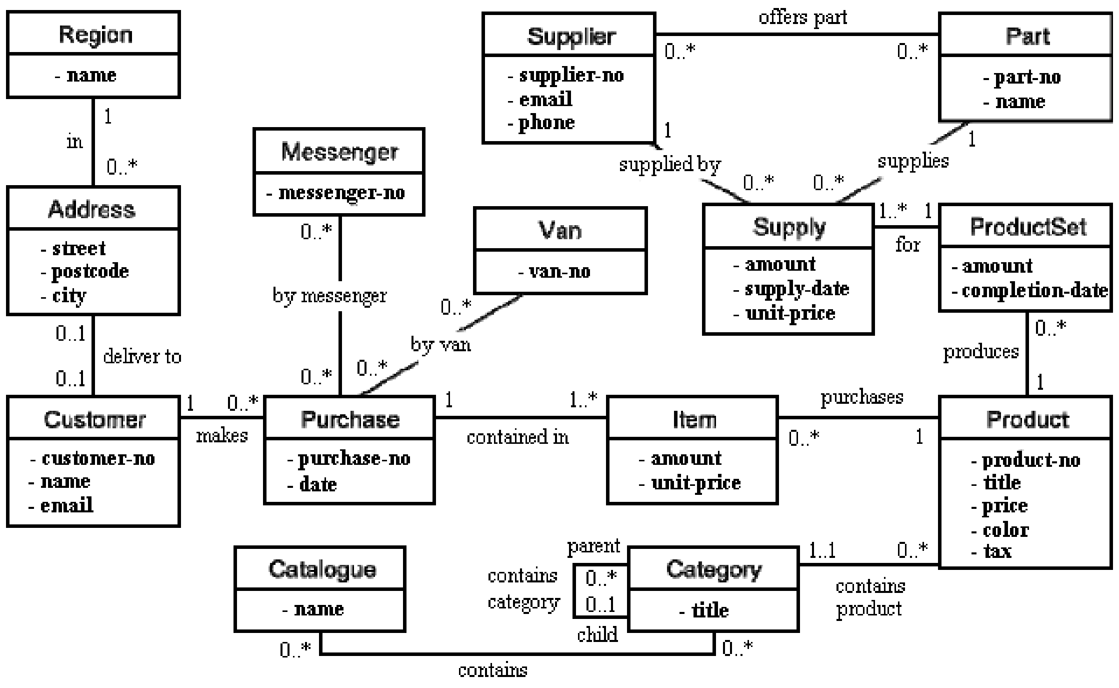


Figure 2.1: Sample XSEM-ER Schema

$S_1, S_2, \ldots, S_n$. It means that an instance of a specific entity type S_i is also an instance of G and has values of its attributes.

The fact that S_i is a specific entity type to a general entity type G can be written $(S_i, G)_{IS-A}$.

Unlike other models that extend the E-R model, XSEM-ER doesn't contain new modeling constructs applying the special features of XML. The only extending modeling construct is called a **cluster type**. It should be applied for modeling irregular data on the conceptual level.

For example, for the modeled data, there can be a requirement that a purchase order is assisted by a sales person or a web application. Both possibilities must be expressed.

A cluster type is a solution, because it can group two or more entity types. It can be assigned as a participant to a relationship type or a determinant to a weak entity type. Even entity types with no common semantics can be grouped by a cluster type.

In Figure 2.1 there is a sample XSEM-ER schema. In this text, we describe the ways of displaying the PIM objects just in words. Their illustrations can be found on many XSEM-ER examples in [4].

An entity type is displayed by a rectangle, containing the name of the entity type and list of its attributes (datatypes are not shown). If the entity type is weak, moreover, an inner hexagon is added.

A relationship type is displayed by a hexagon, again, with the name and the attributes listed. The participants are connected with the hexagon by a solid arrow oriented from the relationship type to the participant.

An IS-A relationship type is displayed by an empty arrow oriented from the special to the general entity type. A cluster type is represented by a small circle with a symbol + in it.

2.1.2 XSEM-H

XSEM-H is a platform-specific model (PSM) from the MDA point of view. It expresses how data modeled by XSEM-ER models are represented in XML documents.

Schemas expressed in the XSEM-H model are called XSEM-H views. Each XSEM-H view is a graph that models a given type of XML documents. Its nodes are formed by the entity and relationship types, defined in XSEM-ER models. Its edges are formed by so called hiararchical projections of node types.

The **hierarchical projection** of a relationship or a weak entity type is an expression specifying a required hierarchical representation of a given type in a formal way. Each hierarchical projection consists of two entity or relationship types called parent and child, and a sequence of entity or relationship types, which is called a context.

An entity or relationship type from an XSEM-ER schema can be represented two or more times in the same XSEM-H view. Each node in an XSEM-H view can have some attributes, content and element label.

An attribute set of an XSEM-H view node is a subset of all attributes of the entity or the relationship type, which is represented by the node. Each attribute can have its alias assigned. If we want an attribute to model an XML

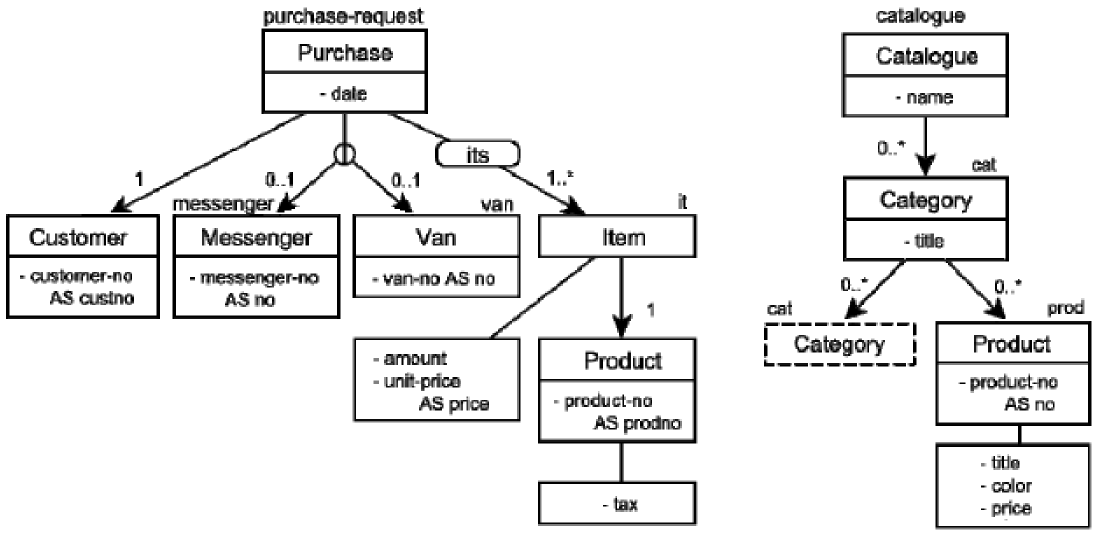


Figure 2.2: Sample XSEM-H View

element instead of an XML attribute, we put it into a special construction called **attribute container**.

A content of a node is an ordered sequence of edges going from the node and attribute containers (and content containers, content choices, see below) assigned to the node.

An element label of a node is optional.

In an XSEM-H view, sometimes, we need to specify that XML elements, modeled by one or more edges in a content of the node, are enclosed into a separate XML element. For that purpose, there is a construction called **content container** in the XSEM-H model. Content container has always its name assigned.

Sometimes, XML documents have an irregular structure, e.g. a purchase order can be made by a new or an existing customer. We must be able to expressed the situation in our XML schema. For modelling such an irregular structure, **content choices** and **node choices** are defined in XSEM-H.

A **structural representative** is a construction that is useful when different nodes of an XSEM-H view represent the same entity or relationship type.

Without using the structural representative we must repeat the same representation several times at different locations in the XSEM-H view. Using the construction, a recursive structure of data can be expressed, as well as IS-A hierarchies described in XSEM-ER.

The detailed description of the XSEM-H model in [4] also focuses on two other features, very typical for XML: an ordering of edges (of a node) and a representation of mixed content.

In Figure 2.2 there are sample XSEM-H views, both are based on the XSEM-ER schema shown in Figure 2.1.

The views are visualized as follows: A node is a rectangle. In the rectangle a name of the node can be found. A list of its attributes is shown under the node name. An element label of the node is displayed above the rectangle. An edge is represented by a solid arrow oriented from the parent to the child node. For each edge the cardinality of the parent node is displayed.

2.2 XCase Tool

In this section, we provide an example of work with a case tool for conceptual modeling for XML called XCase. It implements the conceptual model XSEM described in Section 2.1 and allows user to design XML schemas at the conceptual level in a comfortable way.

The process of XML schema design is separated in two parts, as required by the XSEM model. In XCase, PIM diagrams are realizations of XSEM-ER models, while PSM diagrams are realizations of XSEM-H models.

We want to give you a basic idea, what is the XCase tool good for and how the XSEM model is implemented in it. More detailed description of the application can be found in the oficial XCase User Guide.

2.2.1 PIM/PSM Modeling

We demonstrate work with XCase on an example, on which we can show most of special XSEM constructions easily.

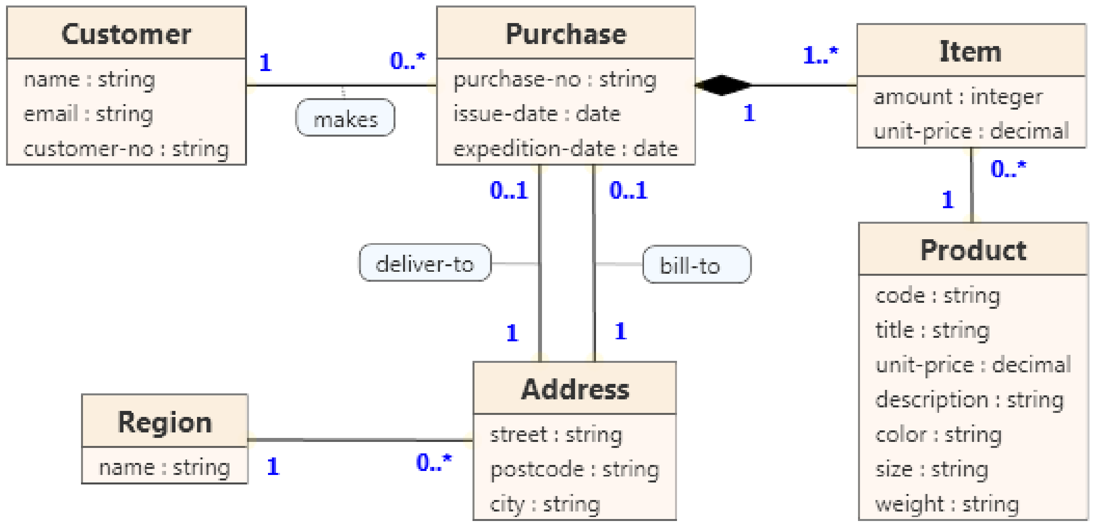


Figure 2.3: Sample PIM Diagram, Sales

We have a company that produces and sells some products. We want to model a purchase request for the company.

When modeling the situation in XCase tool, first, we must make the decision what entity types should be used and what relations the model should contain. With this knowledge we can construct a PIM diagram, illustrated in Figure 2.3.

In Figure 2.3 rectangles with labels *Customer*, *Purchase*, *Item*, *Product*, *Address* and *Region* represent entity types of the XSEM-ER model.

In top parts of the rectangles there are names of the entity types. Attributes of each entity type are displayed under the names. Their names and datatypes are listed.

In this example no relationship types are included. Instead, labels *makes*, *deliver-to* and *bill-to* are added to some associations.

Figure 2.4 illustrates a PSM diagram of a purchase request. It represents an XSEM-H view, as introduced in Section 2.1.2.

The XCase tool won't let you put a node, that is not defined in any PIM diagram, to your PSM diagram. It also checks if the relationships at the PSM level and at the PIM level are mutually consistent.

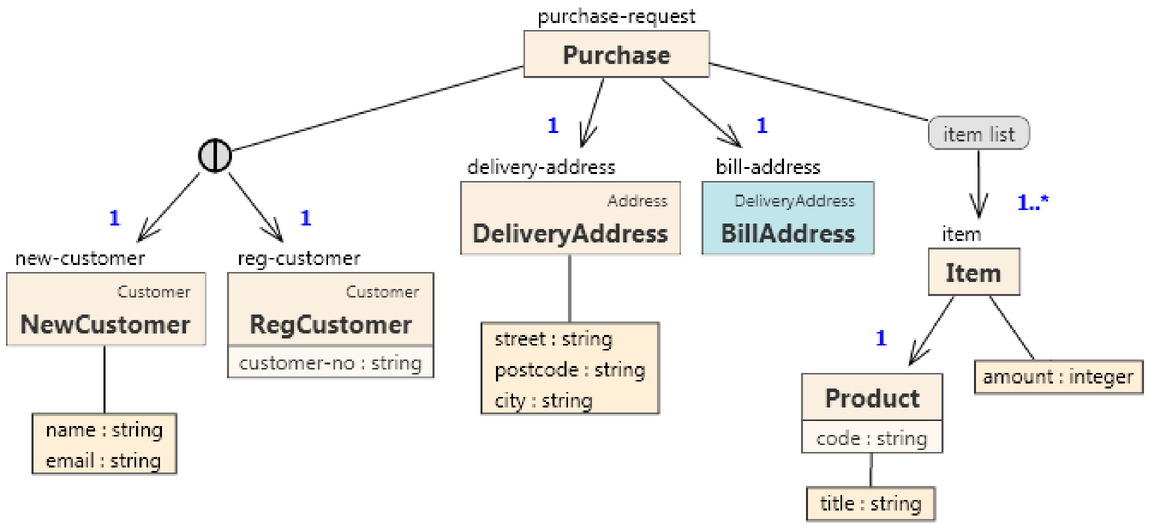


Figure 2.4: Sample PSM Diagram, Purchase Request

Nodes of the PSM diagram, depicted as rectangles, are mainly derived from the entity types of the PIM diagram. The grey node with label *item list* is a content container. The grey circle with vertical line in it is a content choice. Attribute containers are depicted as nameless rectangles with attribute declarations only.

The blue rectangle with label *BillAddress* is a structural representative of the one with label *DeliveryAddress*.

Chapter 3

Basic Translation Algorithm

Modeling data representation on the conceptual level first is very benefical, but for practical reasons we need a possibility to convert an XSEM-H model onto the logical level. For modeling of XML data representation, the logical level is represented by its expression in any XML schema language.

The translation from the conceptual level to the logical one can be done automatically. An example of an algorithm realizing such a translation can be found in [4], where an output of the algorithm is written in the XML Schema language. In this work, let us call it the basic translation algorithm.

In Section 3.1 we focus on brief description of the basic translation algorithm and in Section 3.3 we discuss some of its limitations. The algorithm is presented with a descriptive example of its input and corresponding output. In Section 3.2 a set of all possible outputs of the algorithm is defined formally.

3.1 Brief Description

We describe the basic translation algorithm in brief and demonstrate it on an example. A detailed description of the algorithm can be found in [4]. In Appendix A at the end of this thesis, we summarize the translation of the XSEM-H constructs in a table.

The basic translation algorithm gets an XSEM-H view as its input structure and gives an output in the XML Schema language. For each root node of the view, it creates a global declaration, continues with translation of the node's contents and then proceeds recursively.

Nodes of an XSEM-H view with defined element labels are translated to global *xs:complexType* definitions. Root nodes also add global *xs:element* definitions. Nodes without element labels are translated into model groups and attribute groups.

Each XSEM-H view node has some attributes and contents. Attributes model a set of XML attributes. Contents model a sequences of XML elements and are translated to a *xs:sequence* constructions. Each content is an edge, attribute container, content container, or a content choice.

Edges of an XSEM-H view are translated into element declarations referencing complex types if child of the association is translated into a complex type. If it is translated into groups, references to the groups are propagated to the complex types above.

An attribute container is translated to a sequence of element declarations.

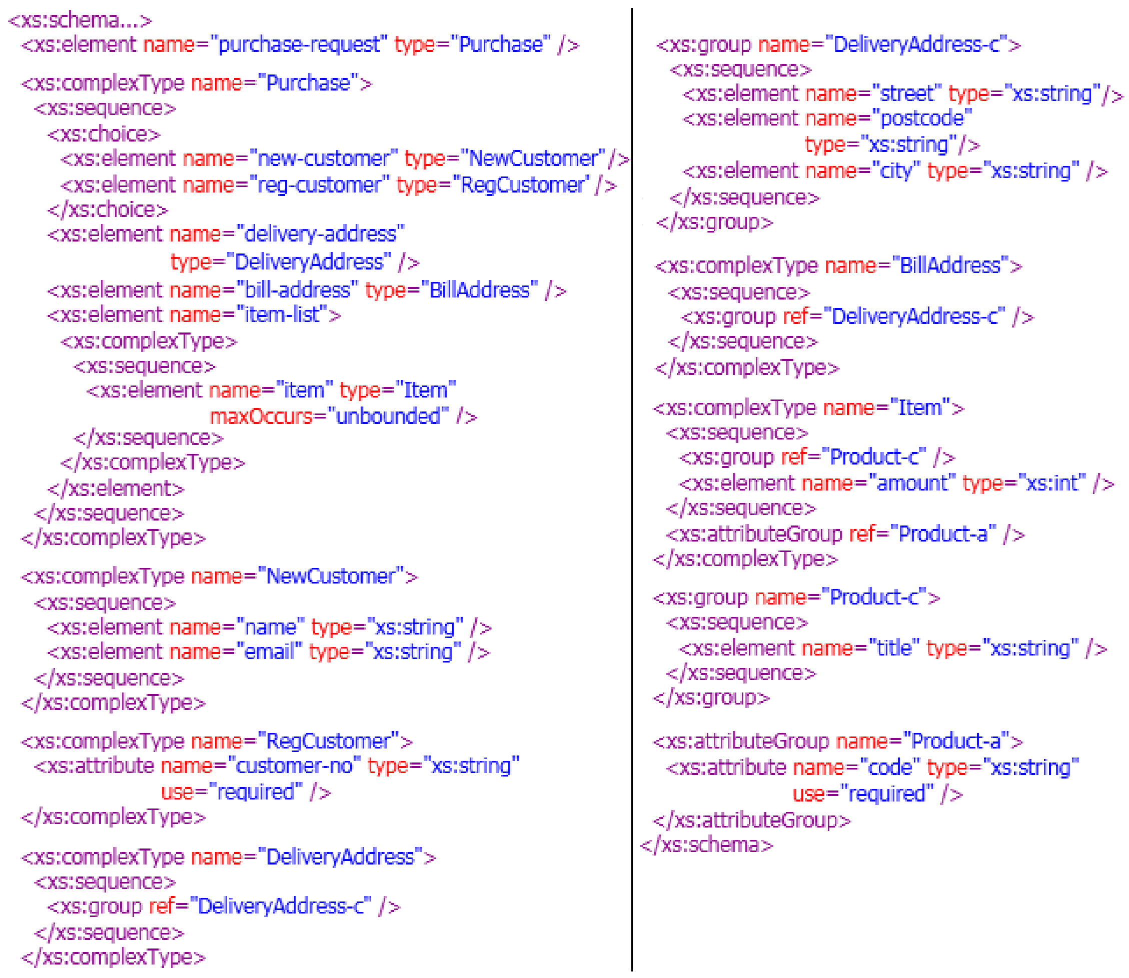
A content container is translated to an element declaration and a complex type declaration, that is a child of the element declaration. A content of the content container is translated to a *xs:sequence* of the complex type declaration.

A content choice is translated to a *xs:choice* content model. Translations of its contents are placed directly in the *xs:choice* construction.

The XSEM-H model defines a structural representative construct, as described in 2.1.2. Let U be a structural representative of V. If V does not have an element label, it is translated to a model and attribute group. If it does have an element label, it's not translated into a complex type, as expected. Instead, it's translated into an attribute group, a model group and a complex type definition. The complex type definition contains references to both the groups.

U itself is translated to a *xs:sequence* content model, that contains a reference to the model group of V, as well as translations of contents of U. Attributes of U are translated to a reference to the attribute group of V and translations of attributes of U.

For each child of U without an element label and not contained in a content container, a translation of attributes of U is extended with the reference to the attribute group translated from the child (if there is any).



```
<xs:schema...>
 <xs:element name="purchase-request" type="Purchase" />

 <xs:complexType name="Purchase">
  <xs:sequence>
   <xs:choice>
    <xs:element name="new-customer" type="NewCustomer"/>
    <xs:element name="reg-customer" type="RegCustomer' />
   </xs:choice>
   <xs:element name="delivery-address"
                  type="DeliveryAddress" />
   <xs:element name="bill-address" type="BillAddress" />
   <xs:element name="item-list">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="item" type="Item"
                   maxOccurs="unbounded" />
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="NewCustomer">
  <xs:sequence>
   <xs:element name="name" type="xs:string" />
   <xs:element name="email" type="xs:string" />
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="RegCustomer">
  <xs:attribute name="customer-no" type="xs:string"
                  use="required" />
 </xs:complexType>

 <xs:complexType name="DeliveryAddress">
  <xs:sequence>
   <xs:group ref="DeliveryAddress-c" />
  </xs:sequence>
 </xs:complexType>

 <xs:group name="DeliveryAddress-c">
  <xs:sequence>
   <xs:element name="street" type="xs:string"/>
   <xs:element name="postcode"
                  type="xs:string"/>
   <xs:element name="city" type="xs:string" />
  </xs:sequence>
 </xs:group>

 <xs:complexType name="BillAddress">
  <xs:sequence>
   <xs:group ref="DeliveryAddress-c" />
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="Item">
  <xs:sequence>
   <xs:group ref="Product-c" />
   <xs:element name="amount" type="xs:int" />
  </xs:sequence>
  <xs:attributeGroup ref="Product-a" />
 </xs:complexType>

 <xs:group name="Product-c">
  <xs:sequence>
   <xs:element name="title" type="xs:string" />
  </xs:sequence>
 </xs:group>

 <xs:attributeGroup name="Product-a">
  <xs:attribute name="code" type="xs:string"
                  use="required" />
 </xs:attributeGroup>
</xs:schema>
```

Figure 3.1: Sample Basic Algorithm Result, Purchase Request

XSEM-ER model allows defining generalizations using special constructs called IS-A hierarchies. The XML Schema language has adequate constructions to allow translations of such relationships. Let U be a node specialized by V_1, . . ., V_n.

If U has an element label, it's translated to a complex type definition. If U is abstract, the complex type definition has the parameter *abstract* set to true. Then, each V_i is translated to a complex type definition containing *xs:complexContent* and *xs:extension* constructs. The *xs:extension* has its *base* set to the name of complex type definition translated from U and contains both a translation of V_i content and a translation of V_i attributes.

If U does not have an element label, it's translated to a model group and attribute group. If V_i does not have an element label as well, it is translated to a model group and attribute group.

Figure 3.1 illustrates an XML schema resulting from basic translation of the XSEM-H view from Figure 2.4.

As you can see, nodes *Purchase*, *NewCustomer*, *RegCustomer*, *DeliveryAddress*, *BillAddress* and *Item* have element labels and therefore they are translated into complex type declarations.

BillAddress is a structural representative of *DeliveryAddress*, therefore the *DeliveryAddress* is translated not only into the complex type declaration, but into the model group *DeliveryAddress-c* as well.

Thanks to that, the content of *DeliveryAddress* can be referenced from *BillAddress* complex type, using a reference to the model group.

There are four attribute containers in the XSEM-H view. Their translations can be found easily, e.g. translation of the attribute container with *name* and *mail* attributes is included in a declaration of the complex type *NewCustomer*.

Content container *item list* results in a local declaration of element *item-list* with local and nameless complex type declaration. Content choice between *NewCustomer* and *RegCustomer* is translated into a *xs:choice* construct, as expected.

3.2 Input and Output Definition

From Section 3.1 we know how the basic translation algorithm works. Now, we try to clear up how its regular output may and may not look like. We create a graphical notion, that will help us in next chapters with description of limitations and proposed improvements of the algorithm.

Also, using PSM diagrams we describe an XML schema against which all the possible outputs of the basic translation algorithm are valid. We refer to these diagrams at least from Chapter 5, where we are looking for types of redundancy that can occur in an arbitrary output of the basic translation algorithm.

Generally, an input of the algorithm is an XSEM-H view. An output is an XML schema expressed in the XML Schema language. When working with an output of the algorithm, we often use terms from the world of XML.

Therefore, if we say that something is a parent or a child of something else, we refer to their relationship in an XML structure of an XSD document.

3.2.1 Space of All XML Schemas

Let us imagine a space of all existing XML schemas and let our image be very abstracted. That means, in our imagination items of the space are not expressed in any XML schema language nor displayed in any model, like *XSEM*.

Definition 1 *XD is a set of all XML documents. XS is a set of all XML schemas. Let $xd_m \in XD$ and $xs_n \in XS$. Then $xd_m \lhd xs_n$ denotes the fact that an XML document xd_m is valid against an XML schema xs_n.*

For every XML schema language, a set of all XML schemas expressible in the language is a subset of XS. Now, we want to know how intersections of all these subsets look like.

Because of both input and output of the algorithm are XML schemas, also, we want to know how sets of all inputs and outputs look like in XS. In the following text we present a graphical idea of how approximately XS is organized.

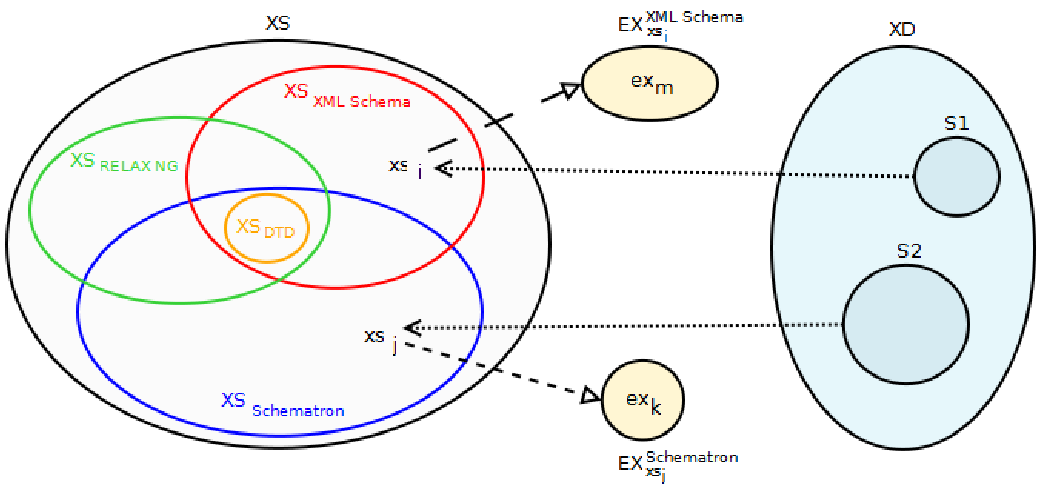


Figure 3.2: The Most Widely Used XML Schema Languages

Definition 2 *Let L be an XML schema language. Then XS_L is a set of all XML schemas expressible in L. Further, $\forall\ xs_i \in XS$*

- *if $xs_i \in XS_L$, $EX^L_{xs_i}$ is a set of all possible expressions of xs_i in L*
- *if $xs_i \notin XS_L$, $EX^L_{xs_i}$ is an empty set*

Figure 3.2 demonstrates Definitions 1 and 2. It shows XS, a space of all existing XML schemas. According to Definition 2, each subset of XS has a name of a particular XML schema language as its subscript and consists of all XML schemas expressible in that language.

We consider only the most widely used languages for XML schema description, i.e. XML Schema [2], RELAX NG [3], DTD [1] and Schematron [17]. For convenience, in Figure 3.2, the subsets are distinguished by color.

Let $xs_1, \ldots, xs_n$ be particular XML schemas. Two of them, xs_i and xs_j are depicted in Figure 3.2. The dashed line is directed from an XML schema to a set of all expressions of the schema in a specified language. The dotted line from XD towards the schema represents a fact that the schema describes a set of XML documents, from which the arrow heads.

$$\forall\ xd_m \in S1 : xd_m \triangleleft xs_i$$

In the following text we consider the XML Schema language (the red one) only, because the basic algorithm cannot give an output in another language. The other languages are shown in Figure 3.2 just for giving the reader the right context. Therefore, you won't see them in next figures. For the XML Schema language let us define a synonym *XSch*. Thanks to the synonym next formulas and figures are more transparent.

$$XML\ Schema \sim XSch$$

We already know which XML schemas are expressible in the XML Schema language - simply those within the red area. That means, the basic algorithm cannot give as its output an XML schema, that doesn't belong to the red area.

Now we wonder where an area of XML schemas, expressible in the conceptual model XSEM-H, lies. That means, we are looking for graphical expression of a set of all the possible inputs of the algorithm. We call the set with a symbol XS_{XSEM} and we mark it with a green color in next figures.

Important fact to realize is that translation algorithm can work correctly only for XML schemas belonging into both the areas, i.e. for XML schemas from $XS_{XSch} \cap XS_{XSEM}$.

If XS_{XSEM} was a subset of XS_{XSch}, the algorithm would be perfect, because for every input XSEM-H view there would be no loss of information during the translation. Unfortunately, this is not true, because some constructions of the XSEM-H model cannot be converted into the XML Schema language.

For example, in XSEM-H view we can model the situation that element of an XML document can have either attribute A or attribute B. Such a choice between attributes cannot be expressed in the XML Schema language.

The XS_{XSEM} area must certainly lie within XS, which is the space of all XML schemas. The XS_{XSEM} area must also be at least partly involved within the XS_{XSch}, which means that translation algorithm gives an output for at least one XSEM-H view. This is obvious from the very definition of the algorithm.

At the same time the XS_{XSEM} region acts outside the XS_{XSch}, because not all XSEM-H models can be converted to the XML Schema language without losing an information, as said already.

Figure 3.3 provides a graphical representation of the XS$_{XSEM}$ area. It is divided into subareas A and B. All XML schemas, expressible both in XSEM-H

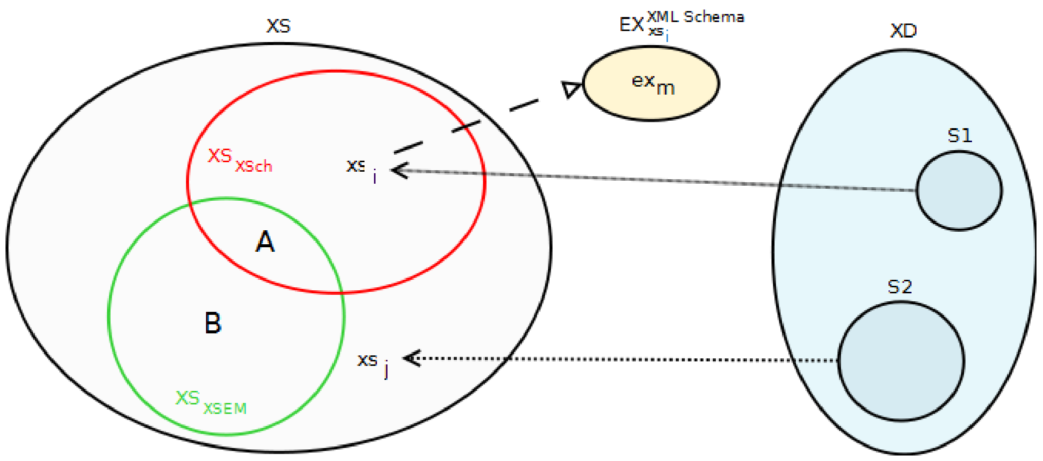


Figure 3.3: XML Schemas Expressible in XSEM-H Model

model and XML Schema language, reside in A. Those which cannot be expressed in the XML Schema language can be found in B.

3.2.2 Algorithm Illustration

The algorithm performing the translation from XSEM-H model into the XML Schema language is a function, assigning exactly one item of $EX_{xs_m}^{XSch}$ to each item of $EX_{xs_i}^{XSEM}$, where xs_i is any item of XS_{XSEM} and xs_m is an item of A. For better illustration, let us introduce some examples.

We want to describe a structure of a certain set of XML documents, that means we need to create an XML schema, against which all documents from the set are valid. Let us suppose that xs_i from Figure 3.4 is the required XML schema. Using a conceptual modeling we create an XSEM-H view of xs_i. In Figure 3.4 the view is represented by one of four rectangles in $EX_{xs_i}^{XSEM}$.

When we run the basic algorithm, the XSEM-H view is converted to an item of $EX_{xs_i}^{XSch}$. Such an item is an XML schema expressed in the XML Schema language. In Figure 3.4 this conversion is expressed with blue arrow.

As a second example, we want to perform the translation of XML schema xs_o.

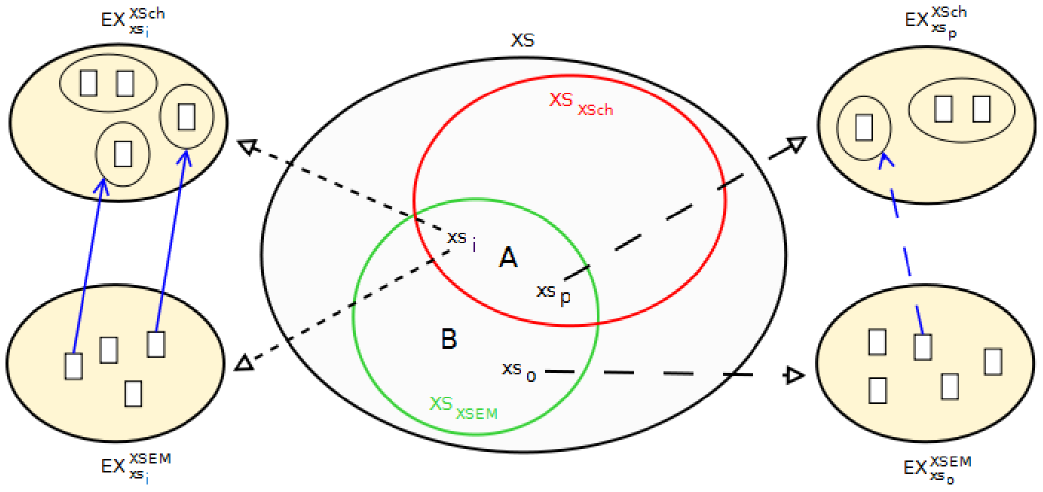


Figure 3.4: Basic Translation Algorithm Process

The problem is that no expression of xs_o exists in the XML Schema language, as it doesn't belong to the red circle. If translation of xs_o shall be performed anyway, some loss of information will be neccessary. In that case, the algorithm changes the XML schema automatically before the whole translation process starts, and then XML Schema output corresponds to the XML schema xs_p, that describes a slightly different structure of XML documents than xs_o.

3.2.3 XML Schema of Output

For each XML schema xs_i from $XS_{XSch} \cap XS_{XSEM}$ the algorithm translates its XSEM-H expression to an expression in the XML Schema language. However, it doesn't use all existing constructions of that language. In this section we describe which parts of the XML Schema language the algorithm uses.

We want to find out not only which XML Schema elements the algorithm uses. Also, we want to know how those elements can be arranged in an XML structure of an output of the algorithm.

In other words, we are looking for a schema against which an output of the algorithm is valid, regardless of which input the algorithm gets. We call it xs_{OUT} in this text.

Obviously, $xs_{OUT} \in XS$ because each XML schema expression in the XML Schema language is a set of XML documents, interconnected by *import* and *include* XML Schema constructions. Formally

$$\forall\, xs_i \in XS_{XSch} \;\forall\, ex_k \in EX_{xs_i}^{XSch} : ex_k \subseteq XD$$

Note: Moreover, if the XML Schema expression ex_k of an XML schema was created by the basic translation algorithm, then $ex_k \in XD$. That's because the basic algorithm always generates only one XML document. It doesn't use *import* and *include* constructions.

We express xs_{OUT} in a form of PSM diagrams, created in the XCase tool. The diagrams are depicted in Figures 3.5 - 3.8.

Furthermore, we bring short comparison of xs_{OUT} to the XML Schema of the whole XML Schema language called xs_{XSch}. Thanks to the comparison, we can see which constructions of the language are not used by the basic translation algorithm.

The XML schema xs_{XSch}, describing the whole XML Schema language, declares 42 XML elements. They are listed below. Those, that are declared in xs_{OUT} as well, are written in blue.

`schema, element, group, attribute, attributeGroup, sequence, complexContent, extension, complexType, choice, anyAttribute, simpleType, restriction, simpleContent, all, annotation, include, redefine, import, selector, field, unique, key, keyref, notation, appinfo, documentation, list, union, minExclusive, minInclusive, maxExclusive, maxInclusive, totalDigits, fractionDigits, length, minLength, maxLength, enumeration, whiteSpace, pattern, any`

The conceptual model XSEM has no constructions to express user defined simple datatypes, therefore, xs_{OUT} cannot define XML elements for description of new simple types, as xs_{XSch} does. That's why *simpleContent*, *restriction* and *simpleType* elements and elements from *list* to *pattern* (including both) cannot be written in blue.

An output of the basic translation algorithm has always *schema* as its root XML element. As illustrated in Figure 3.5 there can be any number of XML

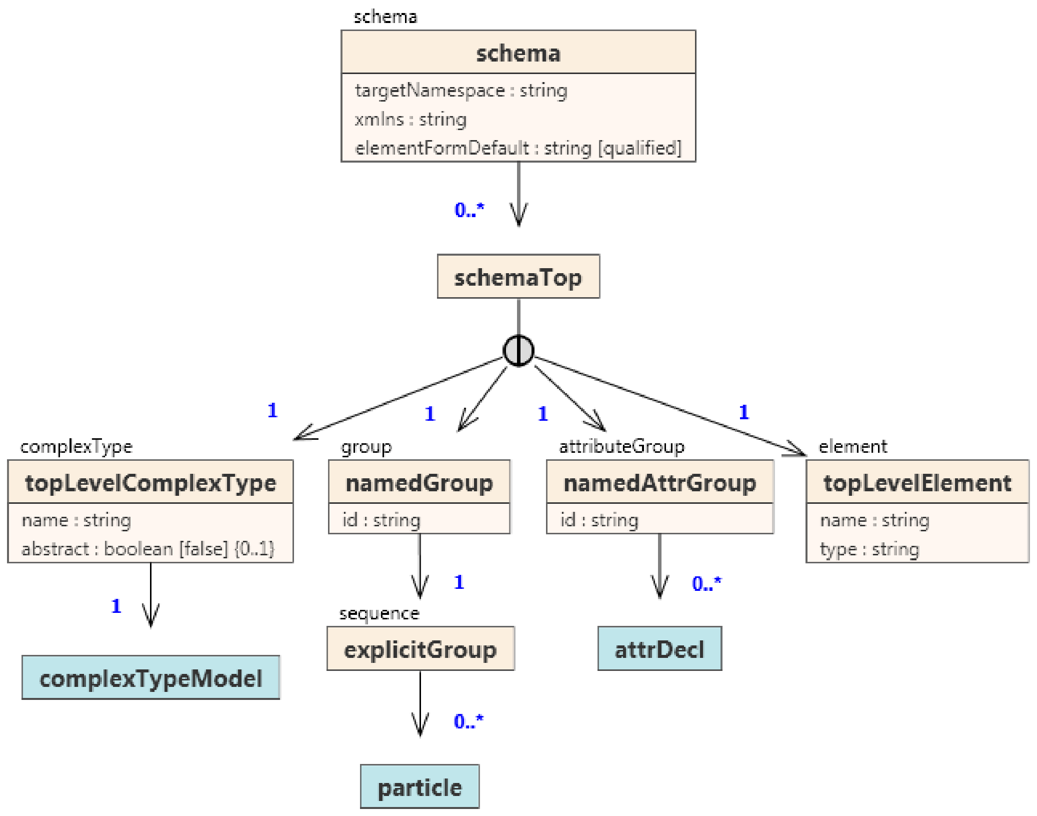


Figure 3.5: Translation Result XML Schema, Top Part

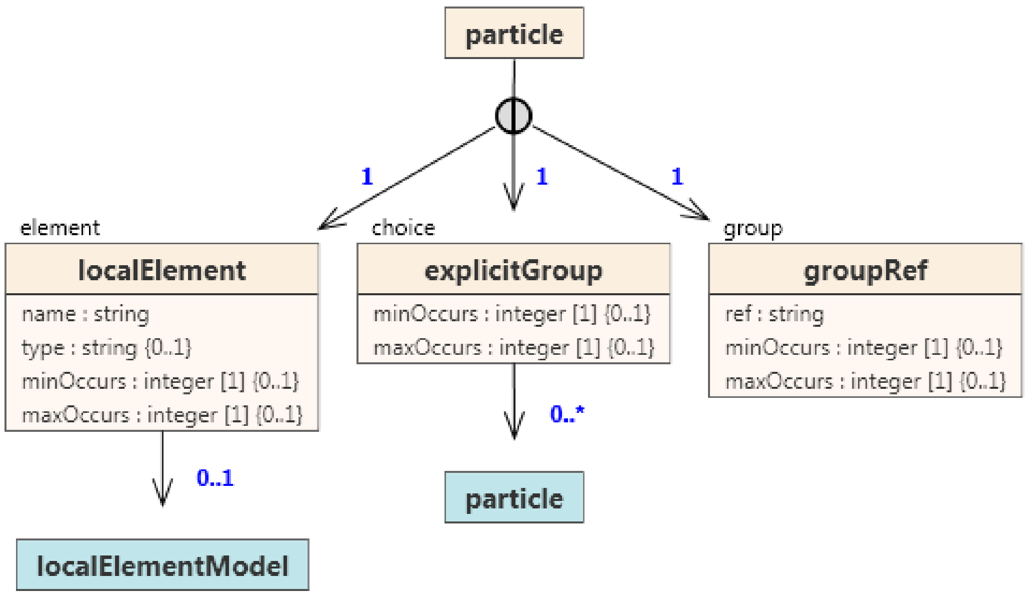

particle
1
1
1
element
choice
group
localElement
name : string
type : string {0..1}
minOccurs : integer [1] {0..1}
maxOccurs : integer [1] {0..1}
explicitGroup
minOccurs : integer [1] {0..1}
maxOccurs : integer [1] {0..1}
groupRef
ref : string
minOccurs : integer [1] {0..1}
maxOccurs : integer [1] {0..1}
0..1
0..*
particle
localElementModel

Figure 3.6: Translation Result XML Schema, Particle

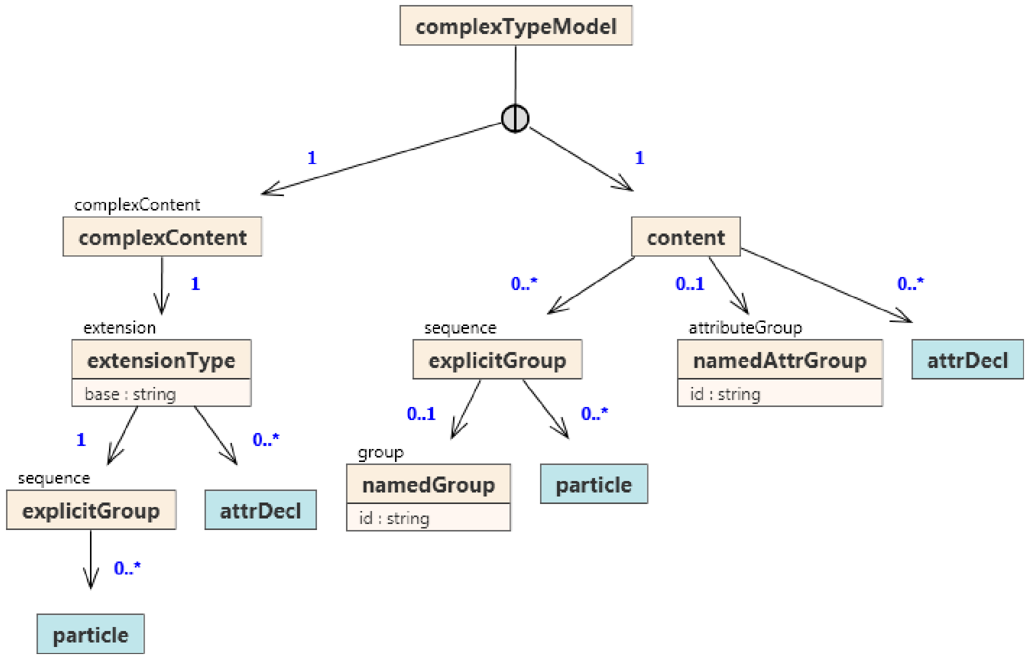


Figure 3.7: Translation Result XML Schema, Complex Type

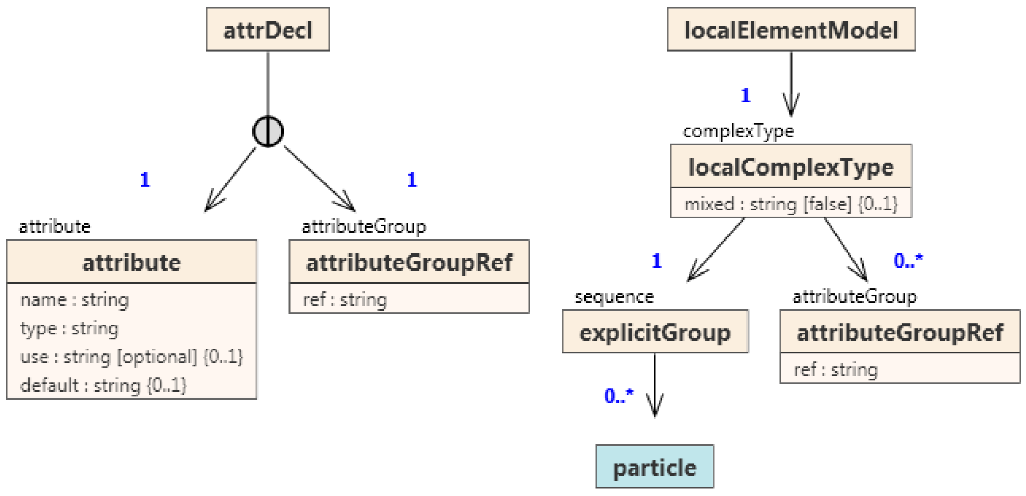


Figure 3.8: Translation Result XML Schema, Others

elements *complexType*, *group*, *attributeGroup* and *element* in the roles of its children.

We can see clearly, which XML attribute declarations are acceptable here. Unlike the possibilities of the XML Schema language there can be no simple type or attribute declaration as a child of the *schema* element.

Obviously, the *group*, *complexType* and *extension* elements can have a *sequence* element as their child. But they cannot have *choice* or *all* elements as their children, in contrast to the situation in the xs_{XSch} XML schema. This restriction of xs_{OUT} is used later in Section 5.3.

Another difference between xs_{OUT} and xs_{XSch} is that in xs_{OUT} *complexContent* element cannot have XML element *restriction* among its children.

In Figures 3.5 - 3.8 some nodes are blue. That is meant to express the fact that content of the node is not declared directly under it, but somewhere else in the figure, or possibly in one of three other PSM diagrams.

Thanks to that the figures are clearer and declarations of nodes are not repeated unnecessarily.

3.3 Limitations

An XSEM-H view, as an input of the basic translation algorithm, is used to describe a structure of a given type of XML documents on the conceptual level. An XML Schema language, as an output of the basic translation algorithm, is used for the same purpose on the logical level.

Both these ways of XML schema expression have their specifics and a good translation algorithm should take advantage of them as much as possible.

In this section, we focus on some limitations and insufficiencies of the basic translation algorithm. However, their solutions are not proposed here, they are discussed later in Section 4.2 and Chapter 5.

Note that our target is not to discuss all existing limitations of the basic translation algorithm. We want to find solutions to just a small subset of them - only the ones, the solution of which is not easy and makes us to propose some special algorithms.

The fact, that the translation algorithm cannot be influenced in its run-time by preset parameters, is not considered to be a limitation. Therefore, we don't discuss it here, but later in Section 4.1, which is devoted to proposed improvements.

3.3.1 Design Patterns

As with software design, there are design patterns associated with XML schema design in the XML Schema language. In this language we have thousands of ways of constructing an XML schema. Choosing the appropriate pattern is a critical step. Once we have made a choice, switching the pattern to another one without specialized GUI tools is inconvenient.

The most popular XML Schema design patterns are Salami Slice, Russian Doll, Venetian Blind [12][13] and Garden of Eden [13].

The basic translation algorithm generates its output in a design pattern, very similar to Venetian Blind. This fact can be limiting for advanced users, e.g. for those who are used to work with XML schemas in Salami Slice design pattern. The algorithm should be able to generate XML schemas in every well-known design pattern and user should have chance to decide which one is suitable for him.

Let us describe in brief the world's most widely used XML Schema design patterns. The description of each design pattern includes a table of its main advantages and disadvantages.

Salami Slice

All elements in the Salami Slice design are global. No nesting of element declarations is required and we can reuse the declarations throughout the XML schema.

The fact that all the elements are global means a greater degree of reusability than Russian Doll and Venetian Blind design patterns. However, this design pattern contains many potential root elements.

Advantages	Disadvantages
Contains all reusable elements.	Exposes the complexity in namespace.
Supports reuse of elements from other documents.	Renders the root difficult to determine.

Russian Doll

The Russian Doll design contains only one single global element. All the other elements are local. We nest element declarations within a single global declaration.

Since it contains only one single global element, the Russian Doll is the simplest pattern to use by instance developers. However, if its types or elements are intended for reuse, the Russian Doll design pattern is not suitable for schema developers.

Advantages	Disadvantages
Contains only one valid root element.	Allows reuse for all or no elements.
Could reduce the complexity of namespace, depending on the elementFormDefault attribute of the schema.	Supports single-file schemas only.

Venetian Blind

The Venetian Blind design pattern contains only one global element. All the other elements are local. We nest element declarations within a single global declaration by means of named complex types and element groups. We can reuse those types and groups throughout the schema.

Venetian Blind is an extension of Russian Doll, in which all the types are defined globally. Because it has only one single root element and all its types are reusable, Venetian Blind is suitable for use by both instance developers and schema developers.

Advantages	Disadvantages
Contains only one single root element.	Limits encapsulation by exposing types.
Allows reuse for all the types and the single global element.	
Allows multiple files.	

Garden of Eden

The Garden of Eden design is a combination of Venetian Blind and Salami Slice design patterns. We define all the elements and types in the global namespace and refer to the elements as required.

Because it exposes all its elements and types globally, Garden of Eden, like Salami Slice, is completely reusable. However, because Garden of Eden exposes multiple elements as global ones, there are many potential root elements.

Advantages	Disadvantages
Allows reuse of both elements and types.	Contains many potential root elements.
Allows multiple files.	Limits encapsulation by exposing types.
	Is difficult to read and understand.

3.3.2 No Redundancy Elimination

The conceptual model XSEM, described in Section 2.1, defines constructs for elimination of redundancy, such as structural representatives or IS-A hierarchies.

However, their use is voluntary and therefore, an arbitrary XSEM-H view often contains a lot of redundancy. In that case, the basic translation algorithm simply transmits the redundancy into its output XML schema, expressed in the XML Schema language.

We want the translation algorithm to be able to eliminate the most common types of redundancy in its output XML schema. When searching for solution of this problem, first, we must find out which types of redundancy really exists in a translation output.

Because we are looking for redundancy on a logical level (XML Schema language), we call it a *structural redundancy*. This is because on the logical level the semantics of data is not expressed and only its structure is important.

A removal of structural redundancy helps to save space, necessary for the representation of XML data, and makes the XML schemas more clear.

To illustrate a structural redundancy in an output of the basic translation algorithm, we choose a very simple example, that is sufficient enough to fully clarify the problem.

In Figure 3.9 there is an XSEM-H view with two nodes and no edge between them. Both the nodes are derived from the same entity type *Animal*, which has at least three simple attributes.

The first node of the XSEM-H view has its element label set to *animal*, the second one to *Animal*. That means, each XML document, matching an XML

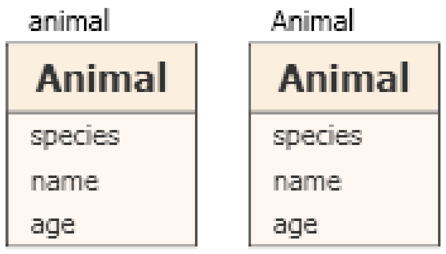
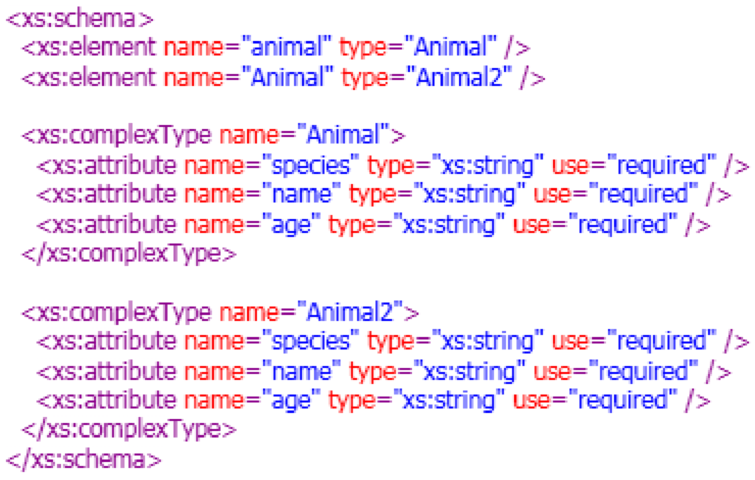


Figure 3.9: XSEM-H View, Example of Redundancy



```
<xs:schema>
  <xs:element name="animal" type="Animal" />
  <xs:element name="Animal" type="Animal2" />

  <xs:complexType name="Animal">
    <xs:attribute name="species" type="xs:string" use="required" />
    <xs:attribute name="name" type="xs:string" use="required" />
    <xs:attribute name="age" type="xs:string" use="required" />
  </xs:complexType>

  <xs:complexType name="Animal2">
    <xs:attribute name="species" type="xs:string" use="required" />
    <xs:attribute name="name" type="xs:string" use="required" />
    <xs:attribute name="age" type="xs:string" use="required" />
  </xs:complexType>
</xs:schema>
```

Figure 3.10: XML Schema, Example of Redundancy

schema represented by this view, must have either *animal* or *Animal* as its root XML element.

At first glance it is obvious that the diagram can be expressed much easier, perhaps with the use of a structural representative construction. In that case, the redundancy would be eliminated at the conceptual level.

In Figure 3.10 there is a result of a basic translation of the XSEM-H view. Obviously, a redundancy was propagated into the XML Schema language level.
We can imagine how the XML schema can be written in the XML Schema language in much better ways. For example, one complex type can be derived

from the other one using XML Schema *xs:extesion* construction.

Or, all the three attribute declarations can be included in a separate attribute group and each complex type can contain a simple references to the attribute group.

The described type of structural redundancy is just one of many. Because, we consider this problem to be very large and important, the whole Chapter 5 is dedicated to it.

3.3.3 Model and Attribute Groups Names

In this section we briefly describe a limitation that is not essential for a translation process. An importance of solving this shortcoming is well below an importance of solving e.g. limitations described in Section 3.3.2 or Section 3.3.4.

If a node of an XSEM-H view doesn't have its element label assigned, the basic translation algorithm places translations of its content into a model group declaration.

Analogously, translations of attributes of the XSEM-H view node are put into an attribute group declaration.

A problem is that values of *name* attributes of model and attribute groups are generated automatically. The automatically generated name for model group always contains name of the XSEM-H view node which the model group is derived from and postfix *-c*. Names of attribute groups begin also with a name of the XSEM-H view node, but their postfixes are set to *-a*.

We consider it right that the group names are derived from the corresponding nodes, but we think the exact form of the names should be determined by user.

Ideally, in each user configuration file there should be a parameter or two, determining a mask or masks for naming groups.

3.3.4 Grouping Entities

In this section we describe a limitation of the basic translation algorithm that might arise if we considered the fact that in real world entities are usually under-

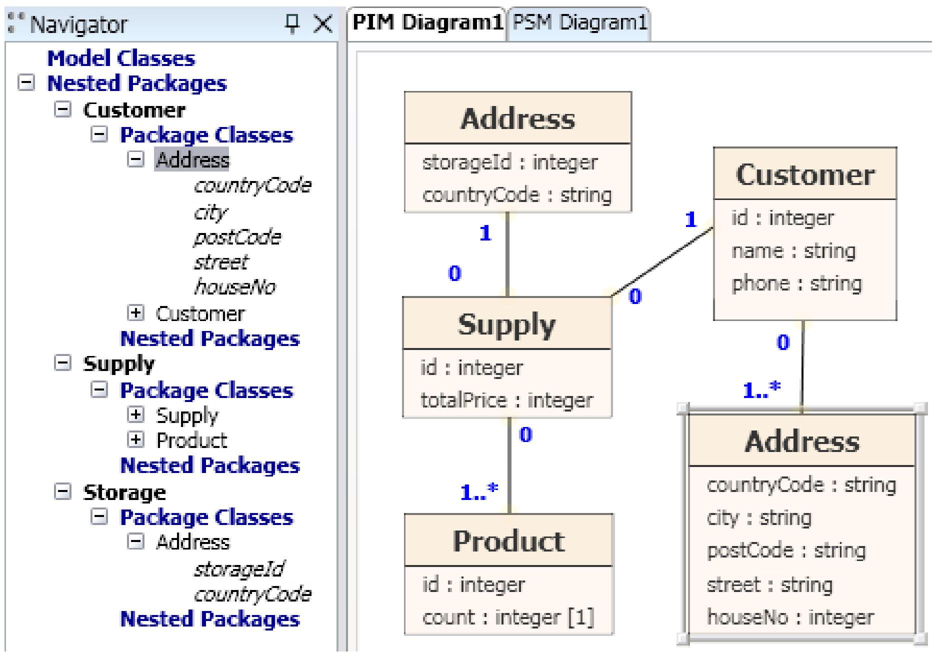


Figure 3.11: XSEM-ER Model, Example of Context Ignoring

stood in any context. However, such an assumption is not explicitly mentioned in the description of the XSEM model.

Let us assume that each entity type belongs to some context, e.g. *cash desk* and *customer* are obviously from the context *shop* or *goalkeeper* and *stadium* are from the context *football*.

Moreover, each entity type can belong to more than one context (e.g. entity type *stadium* to contexts *football*, *rugby*, etc.), but that's not important here, in the problem description.

We want the translation algorithm to be able to propagate the relationships between entity types and their contexts into its resulting XML schema.

Consider a simple project for a small company. In XSEM-ER model we have two entity types with the same name *Address*, as seen in Figure 3.11.

The first entity type represents a place where a customer can be found and belongs to a context called *Customer*. The second is in a context *Storage* and

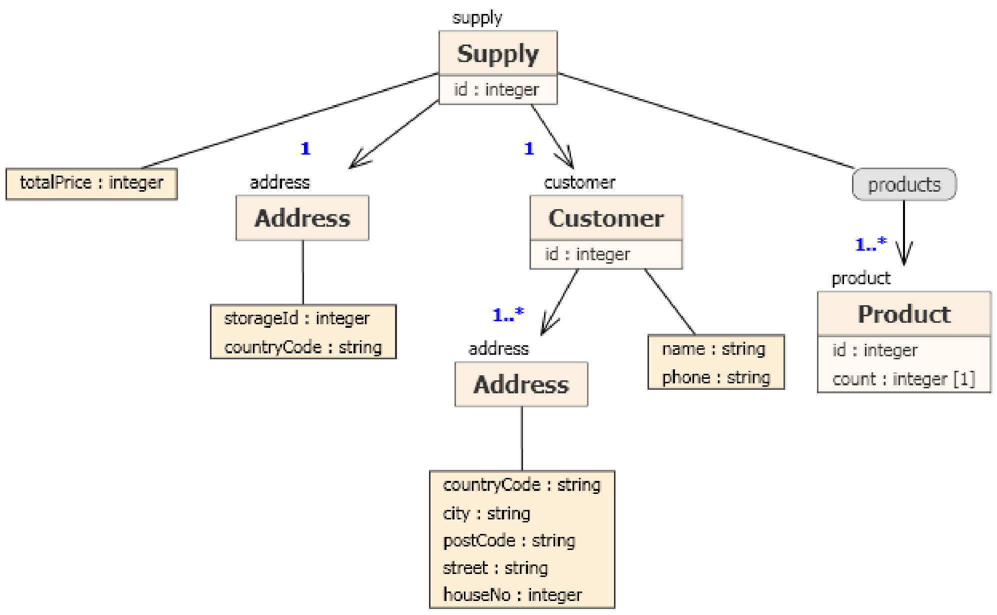


Figure 3.12: XSEM-H View, Example of Context Ignoring

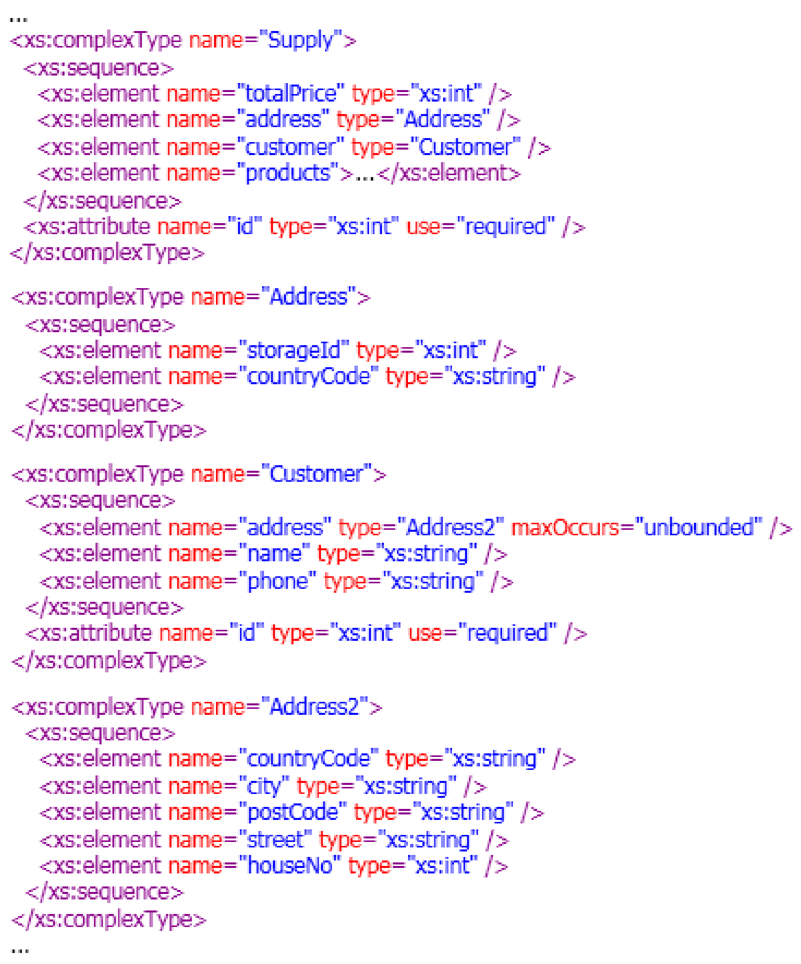
represents an address of a company storage given by a country code and a storage identifier (unique for a specific country).

Furthermore, to illustrate the problem we have an XSEM-H view containing nodes, derived from both the *Address* classes. The view describes a supply of products which already was or is going to be realized. See Figure 3.12.

Supply contains information on its total price, included products and their counts, customer name (with contact info) and information on storage to which products should be delivered. Note that nodes derived from both *Address* classes are included. Their element labels are set to *address*.

If the basic translation algorithm gets the described XSEM-H view as its input, then its output looks like the one in Figure 3.13.

Obviously, if we want to find out which *Address* complex type is derived from entity types, that belongs to context *Storage* and which one comes from entity types, belonging to context *Customer*, we must compare their content properly (and sometimes recursively). Such an approach is very inefficient and we cannot



```
...
<xs:complexType name="Supply">
  <xs:sequence>
    <xs:element name="totalPrice" type="xs:int" />
    <xs:element name="address" type="Address" />
    <xs:element name="customer" type="Customer" />
    <xs:element name="products">...</xs:element>
  </xs:sequence>
  <xs:attribute name="id" type="xs:int" use="required" />
</xs:complexType>

<xs:complexType name="Address">
  <xs:sequence>
    <xs:element name="storageId" type="xs:int" />
    <xs:element name="countryCode" type="xs:string" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Customer">
  <xs:sequence>
    <xs:element name="address" type="Address2" maxOccurs="unbounded" />
    <xs:element name="name" type="xs:string" />
    <xs:element name="phone" type="xs:string" />
  </xs:sequence>
  <xs:attribute name="id" type="xs:int" use="required" />
</xs:complexType>

<xs:complexType name="Address2">
  <xs:sequence>
    <xs:element name="countryCode" type="xs:string" />
    <xs:element name="city" type="xs:string" />
    <xs:element name="postCode" type="xs:string" />
    <xs:element name="street" type="xs:string" />
    <xs:element name="houseNo" type="xs:int" />
  </xs:sequence>
</xs:complexType>
...
```

Figure 3.13: XML Schema, Example of Context Ignoring

be sure that it gives us any solution.

As said already, the problem is that relations between the XSEM-H view nodes and contexts are not reflected in the output of the basic translation algorithm.

In the XML Schema language all global elements, attributes and types must be declared with unique names. By ignoring contexts, sometimes, the basic translation algorithm is forced to make changes to names of some objects.

In our example, the complex type of customer's address got name *Address2*, instead of the expected one: *Address*. The same problem can be simply shown for the names of global elements and attributes.

As an improvement, we wish to modify the basic translation algorithm so that it can meet the following requirements.

- *Correctness* The translation algorithm is not allowed to solve conflicts of names and ids of objects, that are derived from entities of different contexts.
- *Identification* For each object of the resulting schema we are able to easily find the entity group of related XSEM-H view node.
- *Simplicity* The solution is simple and understandable. It doesn't make the resulting XML schema much more complex or less transparent.
- *Location* In the resulting schema, somehow, we are able to see objects, that are generated from the same entity group, together.

Chapter 4

Improving Translation Algorithm

We propose improvements of the basic translation algorithm described in Section 3.1. Most of the improvements refer to limitations of the algorithm that are introduced in Section 3.3.

In sections dedicated to the most important improvements we describe how an output of the algorithm changes with their applications. In that cases we refer to description of an output of the basic translation algorithm, that is introduced in Section 3.2.

Also, we create a concept of translation process profiling, so that anyone, who uses a tool implementing the algorithm, could influence the translation process easily. The concept is based on a configuration. A formal definition of configuration can be found in Section 4.1, as well as an information on how this concept can be implemented.

4.1 Translation Profiling

Designing a concept of translation process profiling is key part of this work. We want to make everyone (who wants to use the algorithm described in Chapter 3) able to create its own profile for translation and also, make him able to re-use it.

We already know that a translation of an XSEM view to the XML Schema language can be done in a big number of different ways - with not the same but equivalent results. With the ability to profile the translation process, users always get the results that best suit their needs. They must just use their

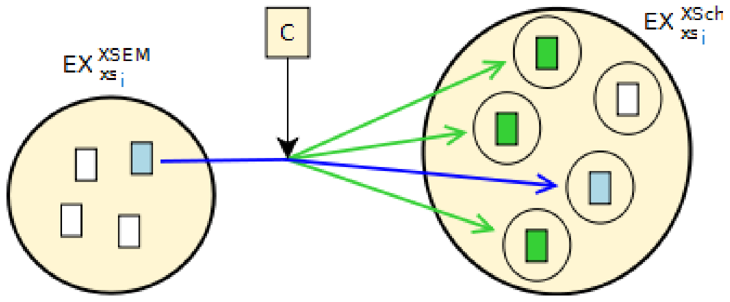


Figure 4.1: Translation Process Profiling

prepared profiles. In this text instead of "profile" we use the term "configuration" or possibly "settings", which all have the same meaning, intuitively.

Definition 3 *A configuration is a set of pairs (k, v) called* parameters*. k is a key and must be unique within the configuration. A set of all possible keys is denoted K. v is the parameter's value. A set of all values that can be used in a pair with key k is denoted V_k. Both keys and values are arbitrary strings.*

$$C = \{(k, v) \mid k \in K \wedge v \in V_k \}$$
$$\forall \ (k_1, v_1), (k_2, v_2) \in C\colon k_1 \neq k_2$$

For now, let us suppose that we know how sets K and V_k look like. In fact, all keys and their possible values are defined in Section 4.2 and Chapter 5.

Among common operations performed with a configuration belong to save it from memory to a persistent storage, to load it again and to get the value belonging to a given key.

In Section 3.2.2 we show a grafical representation of the basic translation algorithm. Because the concept of profiling the translation has a significant influence on the process of the algorithm, it is necessary to show how the graphical representation of the algorithm changes. This is reflected in Figure 4.1.

A blue arrow represents a way how the original translation algorithm works. Square with letter C is a user configuration that is used for sample translation. Green arrows point to new possible outputs. Values of user configuration parameters determine which of the green arrows is used to generate the output.

4.2 Improvement Proposals

Our proposals for improvements are divided into two groups according to their importance and a scope of their description.

The first group contains the only improvements proposal. Its name is redundancy elimination and it is a very complex task. We describe it in a separate chapter - Chapter 5. The second group of improvements proposals contains all the other proposals and they are discussed directly in this section.

If any of the proposals uses the concept of translation profiling introduced in Section 4.1, we indicate which configuration parameters are important for it.

In some cases, we also show how a graphical representation of the basic translation algorithm changes with the proposed improvement.

4.2.1 More Design Patterns

We propose improvement of the basic translation algorithm so that it could generate XML schemas in almost every well-known XML Schema design pattern, specifically Venetian Blind, Salami Slice, Russian Doll and Garden of Eden design pattern. This proposal for improvement is intended to solve the problem described in Section 3.3.1.

The translation algorithm gets an information on which design pattern shall be used from a user configuration, introduced in Definition 3. A relevant parameter of the configuration has a key *design_pattern* and possible values *salami_slice*, *russian_doll*, *venetian_blind* and *garden_of_eden*.

$$(\ design\_pattern,\ v\ ) \in C$$

$$v \in \{\ salami\_slice,\ russian\_doll,\\ venetian\_blind,\ garden\_of\_eden\ \}$$

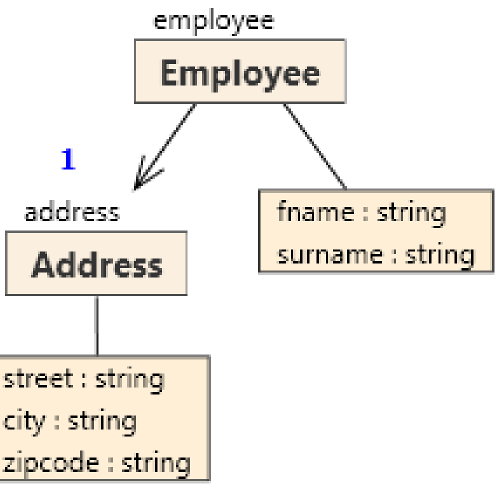
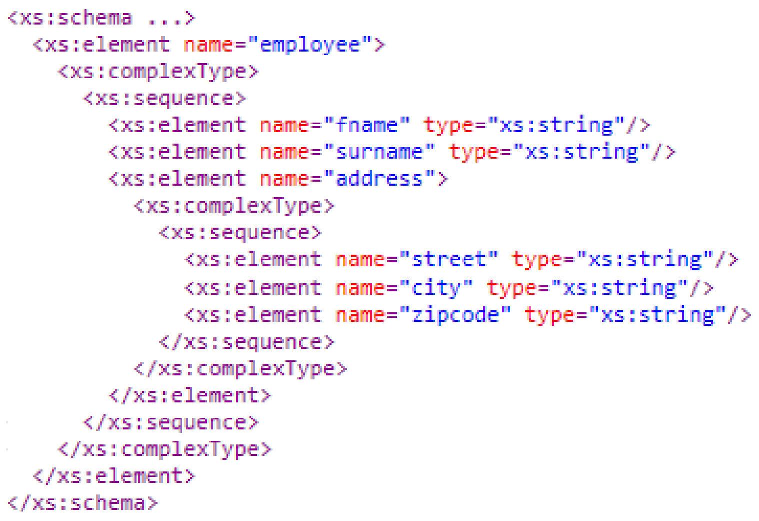
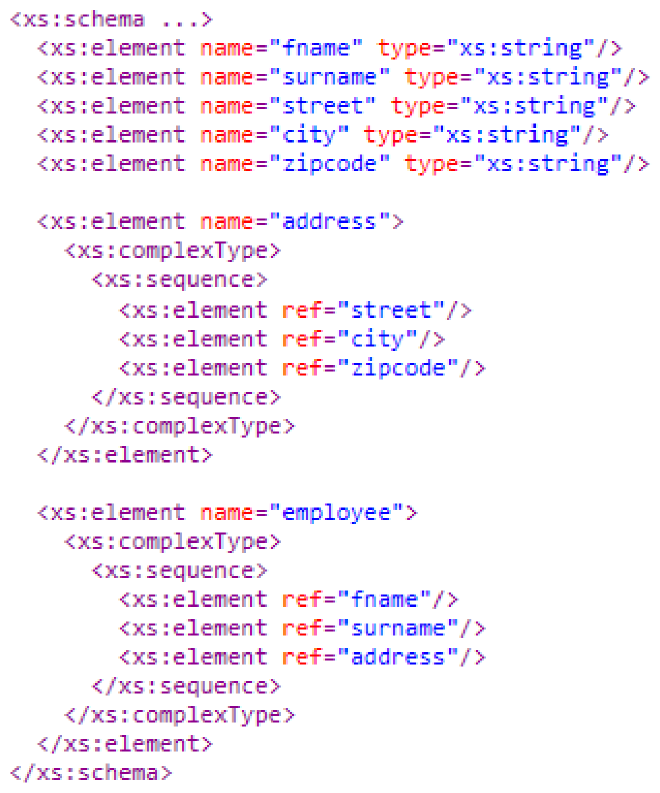


Figure 4.2: XSEM-H View, Design Patterns Improvement



```
<xs:schema ...>
  <xs:element name="employee">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="fname" type="xs:string"/>
        <xs:element name="surname" type="xs:string"/>
        <xs:element name="address">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="street" type="xs:string"/>
              <xs:element name="city" type="xs:string"/>
              <xs:element name="zipcode" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 4.3: XML Schema, Translation into Russian Doll Design



```
<xs:schema ...>
  <xs:element name="fname" type="xs:string"/>
  <xs:element name="surname" type="xs:string"/>
  <xs:element name="street" type="xs:string"/>
  <xs:element name="city" type="xs:string"/>
  <xs:element name="zipcode" type="xs:string"/>

  <xs:element name="address">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="street"/>
        <xs:element ref="city"/>
        <xs:element ref="zipcode"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="employee">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="fname"/>
        <xs:element ref="surname"/>
        <xs:element ref="address"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 4.4: XML Schema, Translation into Salami Slice Design

Regardless of the specified value of the relevant parameter, an improved translation algorithm runs exactly like the basic translation algorithm, except that for each object it decides whether to write it to a global or to a local level. An object is a declaration of an element, attribute, type, group or attribute group, as defined in the XML Schema language. The algorithm also decides whether to place references to global objects on the current level, or not.

To illustrate this improvement proposal on Figure 4.1, in C square there would be a parameter with key *design_pattern* and also, there would be four green arrows pointing to four expressions of an XML schema in the XML Schema language, one each related to one possible value of the parameter.

Let us introduce the proposal with a quite simple example. We have an

XSEM-H view representing employees and their addresses. Both the nodes derived from entity types with the same names have just three attributes and simple element labels. A related XSEM-H view modeled in the XCase tool can be found in Figure 4.2.

In Figure 4.3 we can see how an output of a translation looks like with *design_pattern* parameter set to *russian_doll*. There is just one global XML Schema element declaration and everything else hangs below it.

In Figure 4.4 there is an translation output that we would get if we set *design_pattern* parameter to *salami_slice*. In that case, all elements are global and objects are connected by references.

Analogously, we can show outputs with the parameter set to the remaining values, but the shown figures are sufficient enough to illustrate the proposal.

4.2.2 Naming Masks for Groups

This proposal for improvement is intended to solve the problem described in Section 3.3.3. Its solution is very simple.

We propose to add a parameter to the user configuration, determining a mask for creation of attribute group names. Analogously, we propose a similar parameter for model group names.

$$(\ group\_name\_mask,\ v\ ) \in C$$
$$(\ attrgroup\_name\_mask,\ v\ ) \in C$$

A user can set these parameters to any values that contain exactly one % character. Each attribute group that is created during the translation process gets a value of parameter *attrgroup_name_mask* as its name, only the % sign will be replaced by a name of processed XSEM-H node.

Again, the same rule can be applied for a translation of model groups and the configuration parameter *group_name_mask*.

4.2.3 Considering Entity Groups

In Section 3.3.4 we are given a problem, which can occur if we consider a grouping of real-world entities on the conceptual level of XML schema modeling.

The problem is that from an output of the basic translation algorithm we are not able to find out, which complex type or model group corresponds to which group of entity types.

At the conceptual level, the fact that two entities belong to the same group means, they have something in common. As well we can say that those entities are defined in the same context. By ignoring the grouping of entities during translation, we loose a context of all the objects in an output of the translation algorithm. Such an approach is undesirable.

In this section we solve the problem. We discuss in which ways we can express the fact that XML Schema object is a member of some object group. We try to find a solution that meets four requirements, formulated in Section 3.3.4.

First we define a configuration parameter determining whether the translation algorithm should solve the problem with grouping entities, or not. The configuration parametr has a key *grouping_entities*.

$$(\textit{grouping\_entities},\ v) \in C$$

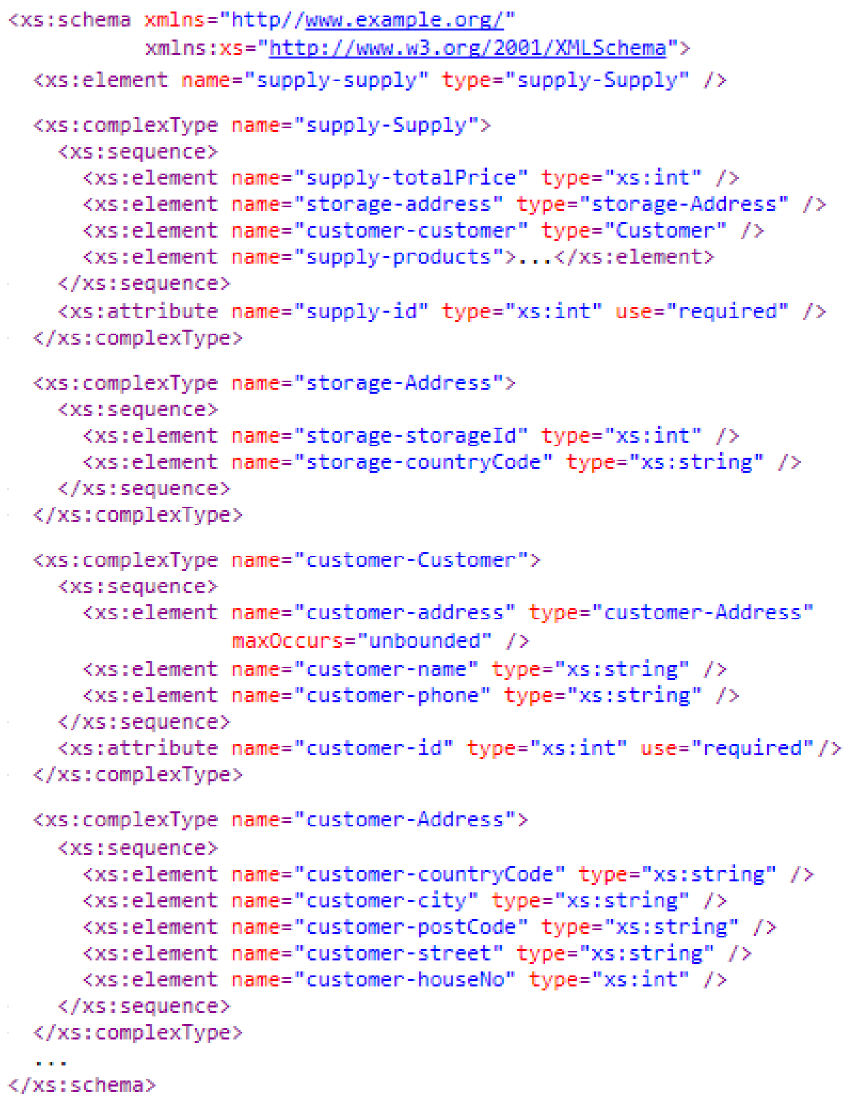
$$v \in \{ \textit{enabled},\ \textit{disabled} \}$$

We discuss two ways of giving a context to elements, attributes and types of the XML Schema language. The first way is quite simple, the second is a complex solution.

The first way is to compose a name of each object from two parts. The first part is a name of the entity group, corresponding to the XSEM class from which the object was created. The second part is an object name that is generated by the basic translation algorithm.

This approach is called the *prefix design*.

An alternative solution is using the well-known concept of XML Namespaces [14]. This approach also uses prefixes for object names. In addition, each prefix has its own namespace assigned that is intended to be (world-wide) unique.



```
<xs:schema xmlns="http//www.example.org/"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="supply-supply" type="supply-Supply" />

  <xs:complexType name="supply-Supply">
    <xs:sequence>
      <xs:element name="supply-totalPrice" type="xs:int" />
      <xs:element name="storage-address" type="storage-Address" />
      <xs:element name="customer-customer" type="Customer" />
      <xs:element name="supply-products">...</xs:element>
    </xs:sequence>
    <xs:attribute name="supply-id" type="xs:int" use="required" />
  </xs:complexType>

  <xs:complexType name="storage-Address">
    <xs:sequence>
      <xs:element name="storage-storageId" type="xs:int" />
      <xs:element name="storage-countryCode" type="xs:string" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="customer-Customer">
    <xs:sequence>
      <xs:element name="customer-address" type="customer-Address"
                  maxOccurs="unbounded" />
      <xs:element name="customer-name" type="xs:string" />
      <xs:element name="customer-phone" type="xs:string" />
    </xs:sequence>
    <xs:attribute name="customer-id" type="xs:int" use="required"/>
  </xs:complexType>

  <xs:complexType name="customer-Address">
    <xs:sequence>
      <xs:element name="customer-countryCode" type="xs:string" />
      <xs:element name="customer-city" type="xs:string" />
      <xs:element name="customer-postCode" type="xs:string" />
      <xs:element name="customer-street" type="xs:string" />
      <xs:element name="customer-houseNo" type="xs:int" />
    </xs:sequence>
  </xs:complexType>
  ...
</xs:schema>
```

Figure 4.5: XML Schema, Grouping Using Prefix Design

We call this solution the *XML Namespace design.*

In Figure 4.5, there is an XML schema resulting from a translation of the XSEM-H view, which was introduced in Figure 3.12. In the translation result the *prefix design* for object names and identifiers is applied.

If we compare the figure to the original XSEM-H view, we can notice that class *Address* from entity group (package) *Storage* resulted in a declaration of XML element and complex type *storage-address.* In contrast, the class *Address* from package *Customer* resulted in declaration of an element and a complex type, both called *customer-Address.*

In Figures 4.6, 4.7 and 4.8, there are three XML Schema files and again, they are formed by a translation of the XSEM-H view from Figure 3.12. In contrast to Figure 4.5 this result implements the *XML Namespace design*, as a solution of our problem.

Obviously, for each non-empty XSEM entity group there is one XML Schema definition file in this solution. Translation of each XSEM-H view node is written in an appropriate file and those files are interconnected via XML Schema elements *import.*

Solutions Comparison

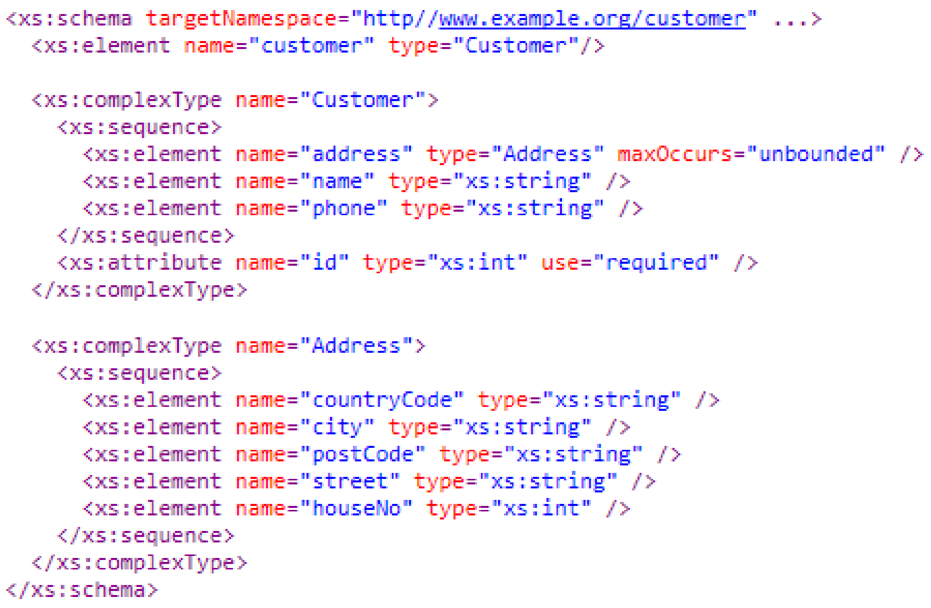
Both the proposed solutions satisfy the *Correctness* and the *Identification* requirement, introduced in 3.3.4.

The *Location* requirement says that we should be able to see all objects, generated from the same entity group, together.

In the *XML Namespace design* such an overview is ensured by a division of XML schema into several XML Schema definition files.

In the *prefix design* we can also create such a division to satisfy the *Location* requirement. In this case, to create links between files, we can use the XML Schema construction *include.*

The assessment of compliance with *Simplicity* requirement is very subjective. Someone may say that the *prefix design* doesn't satisfy it at all, someone else can be satisfied with the clarity of prefix design solutions. In any case, it is clear that the *XML Namespace design* meets the requirement much better, because



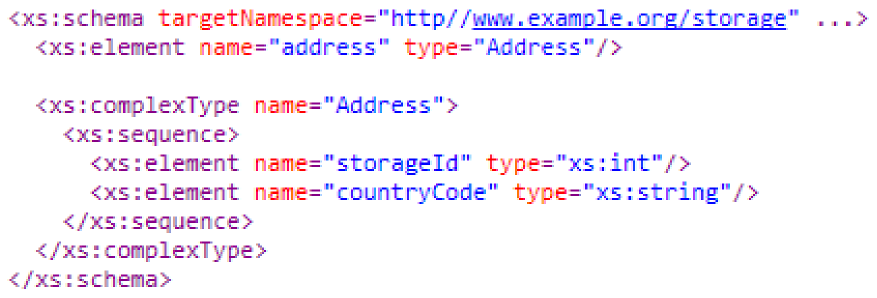
```
<xs:schema targetNamespace="http//www.example.org/customer" ...>
  <xs:element name="customer" type="Customer"/>

  <xs:complexType name="Customer">
    <xs:sequence>
      <xs:element name="address" type="Address" maxOccurs="unbounded" />
      <xs:element name="name" type="xs:string" />
      <xs:element name="phone" type="xs:string" />
    </xs:sequence>
    <xs:attribute name="id" type="xs:int" use="required" />
  </xs:complexType>

  <xs:complexType name="Address">
    <xs:sequence>
      <xs:element name="countryCode" type="xs:string" />
      <xs:element name="city" type="xs:string" />
      <xs:element name="postCode" type="xs:string" />
      <xs:element name="street" type="xs:string" />
      <xs:element name="houseNo" type="xs:int" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Figure 4.6: XML Schema, Grouping Using XML Namespaces, Customer

object names are shorter and simpler.

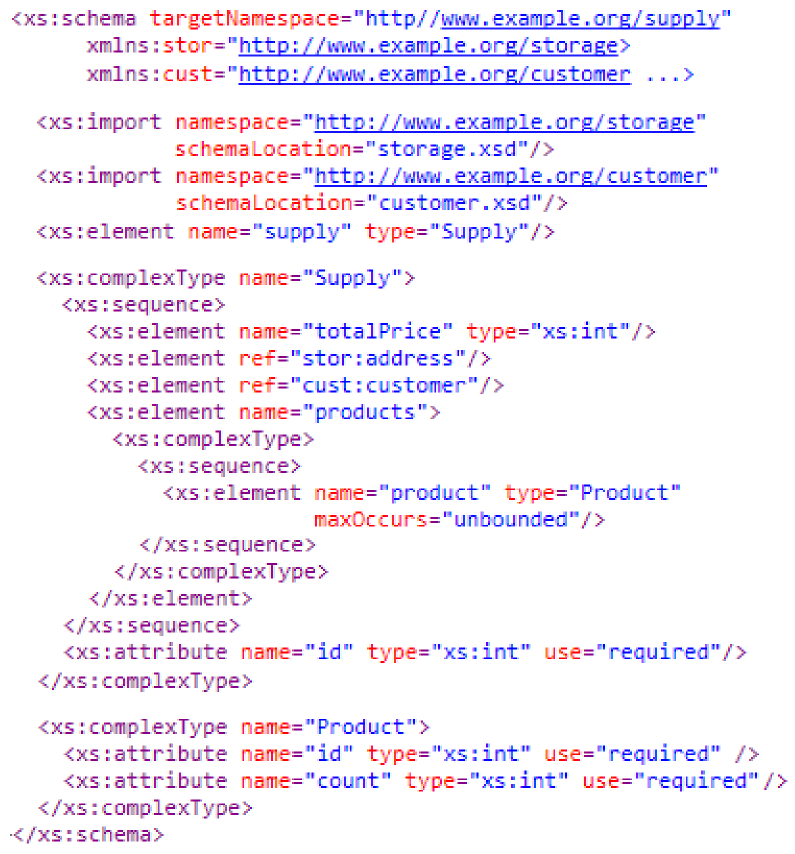
Actually, for the *prefix design* there are two conflicting requirements.
First, the prefix name should be representative, so that there was no problem to work with XML Schema objects, translated from hundreds or thousands of entities groups. Therefore, prefixes must be very long.
Second, each prefix should be short, because it's necessary to repeat it for many objects of XML schema and that is very space-consuming.

Also, the *prefix design* brings a complication, if we want translation results to be automatically processed by specialized applications (XML parsers). An application for work with XML don't expect information on contexts to be hidden in XML schema objects names. Moreover, they have no rule to extract such information (exact context prefix length, delimiter of context prefix and object name, or something like that).
In contrast, all libraries to work with XML can perfectly work with XML Namespaces.



```
<xs:schema targetNamespace="http//www.example.org/storage" ...>
  <xs:element name="address" type="Address"/>

  <xs:complexType name="Address">
    <xs:sequence>
      <xs:element name="storageId" type="xs:int"/>
      <xs:element name="countryCode" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Figure 4.7: XML Schema, Grouping Using XML Namespaces, Storage



```
<xs:schema targetNamespace="http//www.example.org/supply"
      xmlns:stor="http://www.example.org/storage>
      xmlns:cust="http://www.example.org/customer ...>

  <xs:import namespace="http://www.example.org/storage"
             schemaLocation="storage.xsd"/>
  <xs:import namespace="http://www.example.org/customer"
             schemaLocation="customer.xsd"/>
  <xs:element name="supply" type="Supply"/>

  <xs:complexType name="Supply">
    <xs:sequence>
      <xs:element name="totalPrice" type="xs:int"/>
      <xs:element ref="stor:address"/>
      <xs:element ref="cust:customer"/>
      <xs:element name="products">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="product" type="Product"
                        maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="id" type="xs:int" use="required"/>
  </xs:complexType>

  <xs:complexType name="Product">
    <xs:attribute name="id" type="xs:int" use="required" />
    <xs:attribute name="count" type="xs:int" use="required"/>
  </xs:complexType>
</xs:schema>
```

Figure 4.8: XML Schema, Grouping Using XML Namespaces, Supply

An XML schema implementing the *prefix design* may be a complication for instance developers, because each XML document must have contexts in names of all elements to be valid against the XML schema. On the other hand, the *XML Namespace design* force the instance developers to deal with XML Namespaces.

Obviously, the *prefix design* is useful for small XML schemas only. In general, the *XML Namespaces design* is a better solution of the problem with grouping entities.

Proposed Solution

The algorithm uses the concept of XML Namespaces. There is one namespace to every entity group that has its representative in translated XSEM-H view. Each namespace has its own XML Schema definition file.

There is one extra XSD file, that imports definitions of the XSEM-H view root elements from their namespaces. All instances of the XML schema are validated against this extra XSD file. Filenames of XSD files are derived from corresponding target namespace names.

The namespace URIs are in URL form, they are absolute and derived from a name of the XSEM-H view (it there is any) and the names of corresponding entity groups.

Each global declarations of an element, a complex type, a model group or an attribute group is located in XSD with target namespace corresponding to entity group of the class, it was derived from.

Chapter 5

Redundancy Elimination

In Section 3.3.2 we describe the problem of lack of structural redundancy elimination in an output of the basic translation algorithm. In this chapter, we introduce a solution to this problem. We discuss most common types of redundancy that may occur in an instance of xs_{OUT} and propose algorithms for their discovery and removal.

In some cases, users have different opinions on which constructions are redundant. We should be able to eliminate a redundancy completely or to perform the elimination to a level specified by the user in his or her profile. Therefore, proposed algorithms for redundancy elimination are based on the concept of translation profiling, introduced in Section 4.1.

In this work, we deal with three types of structural redundancy. The first type is redundancy in declarations of attributes. The second type is redundancy in declarations of elements. And finally, the third type is redundancy in nestings of XML Schema choice constructions.

For the first type, we create a formal model. We propose two algorithms to eliminate a redundancy in attribute declarations. To deal with the second type, we show how it can be simply transform to the first type of redundancy, if some conditions are satisfied. To eliminate the third type, we propose an algorithm and, of course, introduce it on an explanatory example.

5.1 Redundancy in Attribute Declarations

Both the XML schemas xs_{XSch} and xs_{OUT} introduced in Section 3.2.3 define some attribute and attribute group declarations, as well as referencies to them. In this section we analyze where such declarations and references can be found, how exactly do they look like and which ones are mutually equivalent. We define a new term *attribute part (AP)*. Then, we show how to eliminate redundancies both in attributes and *AP*s.

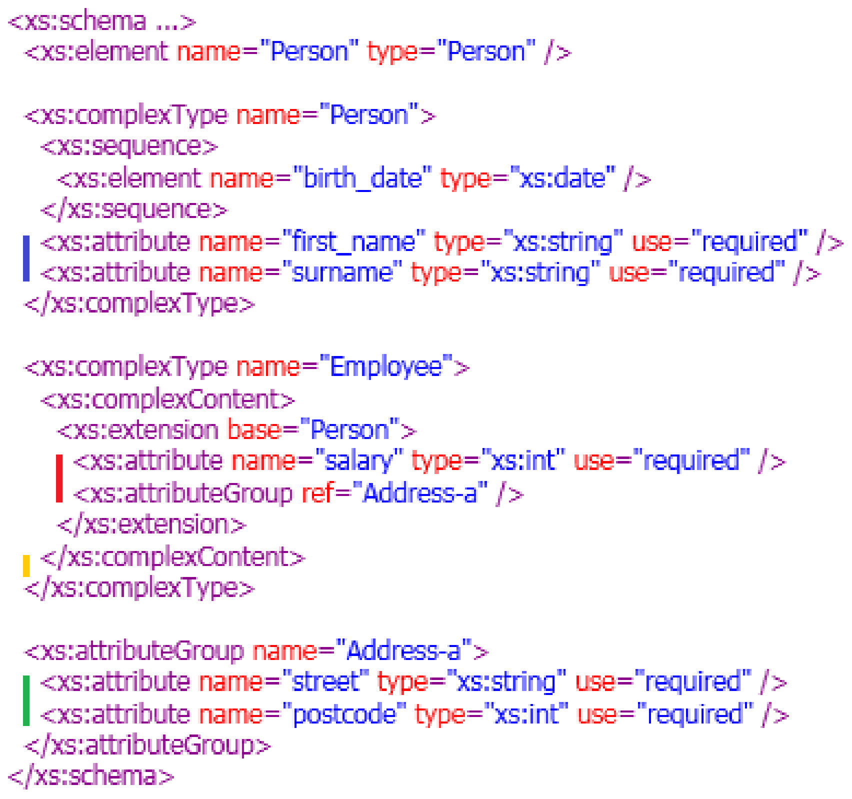
As seen in Figure 3.8, an attribute is declared by an XML element *attribute*. In its *name* attribute we have a name of the declared attribute, type is expressed by a value of its *type* attribute. A default value of the declared attribute can be specified through an XML attribute *default*. In a value of an optional attribute *use* we can specify whether the attribute is *optional*, *required* or *prohibited*.

Unlike the xs_{XSch}, the XML schema xs_{OUT} doesn't allow an attribute declaration in the basic translation output to have a *form*, *fixed* or *id* attribute. Moreover, type of an attribute can not be expressed by a content of XML element *attribute*, as usual in the XML Schema language. And also, in an output of the basic translation algorithm references to attribute declarations can not occur.

As seen in Figure 3.5, in a basic translation output attribute group declarations are formed by an XML element *attributeGroup* with the only XML attribute called *name*. References to the attribute groups are formed by the same element, but instead of *name*, there is an XML attribute called *ref*. That can be seen in Figure 3.8.

Let us explore Figures 3.5 - 3.8 to find out where attribute and attribute group declarations (references respectively) can be expected. Obviously, attributes can be declared under declarations of attribute groups, complex types and in *extension* constructs.

Attribute group declarations can be expected only under a declaration of the whole schema. References to attribute groups can be found under declarations of complex types, another attribute groups or in *extension* constructs.



```
<xs:schema ...>
  <xs:element name="Person" type="Person" />

  <xs:complexType name="Person">
    <xs:sequence>
      <xs:element name="birth_date" type="xs:date" />
    </xs:sequence>
    <xs:attribute name="first_name" type="xs:string" use="required" />
    <xs:attribute name="surname" type="xs:string" use="required" />
  </xs:complexType>

  <xs:complexType name="Employee">
    <xs:complexContent>
      <xs:extension base="Person">
        <xs:attribute name="salary" type="xs:int" use="required" />
        <xs:attributeGroup ref="Address-a" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:attributeGroup name="Address-a">
    <xs:attribute name="street" type="xs:string" use="required" />
    <xs:attribute name="postcode" type="xs:int" use="required" />
  </xs:attributeGroup>
</xs:schema>
```

Figure 5.1: *AP*s Illustration

Definition 4 *Attribute part, denoted AP, is a set of all attribute declarations and references to attribute groups that have the same parent element in an XML structure of a basic translation output such that*

$$\neg\, \exists AP'\colon AP' \supset AP$$

SAP is a set of all nonempty APs in an XML schema.

In Figure 5.1 there is a sample instance of xs_{OUT}, i.e. an output of the basic translation algorithm. According to Definition 4, four APs can be found in it.

A parent of the green AP is an attribute group declaration. The green AP contains two attribute declarations (*street* and *post_code*). The blue AP is under a complex type declaration and contains declarations of two attributes (*first_name* and *surname*).

The red AP hangs under an *extension* contruct and one attribute called *salary* is declared in it. Moreover, one attribute group is referenced from the red AP. The yellow AP is empty.

According to Definition 4, SAP consists of all non-empty APs, i.e. the blue, red and green one.

As already mentioned, an AP can contain some references to attribute groups. Because each of these groups is parent of another AP, we define some relationships between APs.

Definition 5 *$AP_i \in SAP$ is directly dependent on $AP_j \in SAP$, denoted $AP_i \rightarrow AP_j$, when it contains a reference to a parent attribute group of AP_j. AP_i is dependent on AP_j, denoted $AP_i \twoheadrightarrow AP_j$, if there is a sequence $p = p(1), p(2), \ldots, p(n)$ of indices from SAP such that*

$$AP_i \rightarrow AP_{p(1)} \wedge AP_{p(1)} \rightarrow AP_{p(2)} \wedge \ldots \wedge AP_{p(n)} \rightarrow AP_j$$

Obviously, $AP_k \rightarrow AP_m$ implies $AP_k \twoheadrightarrow AP_m$. Figure 5.2 illustrates dependencies between APs from Figure 5.1. The red AP is directly dependent on both the green one ($AP_{red} \rightarrow AP_{green}$) and the blue AP.

Theoretically, we could also define the relationship between AP with an *extension* construct as a parent and AP, whose parent is given by a value of *base* attribute (of the *extension* construct).

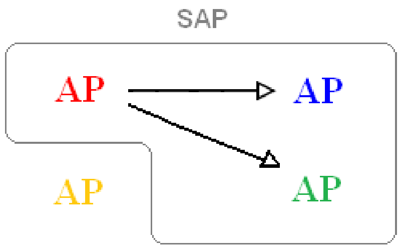


Figure 5.2: APs Relations

In Figure 5.1, such a relationship is between AP_{red} and AP_{blue}.

However, we do not define it, because we don't want to affect ancestor-descendant relationships of complex types at all.

Definition 6 *Let $AP \in SAP$. AD_{AP} is a set of all attribute declarations from AP. AR_{AP} denotes a set of all attribute declarations from both AP and those APs, that are dependent on AP.*

From Definition 6 we derive a relationship between the two sets. AR set of each AP is equal to a unification of its AD set and AR sets of all APs that directly depends on AP. This can be written as follows.

$$AR_{AP} = AD_{AP} \cup \{AR_{B_j} | AP \rightarrow B_j\} \tag{5.1}$$

In Figure 5.2, the APs and their direct dependencies together form a graph. We create a formal definition of such a graph.

Definition 7 *The graph of APs is a directed graph $GAP = (V,E)$, where $V = SAP$ is a set of vertices and $E = \{(v1,v2) \mid v1,\ v2 \in V \land v1 \rightarrow v2\ \}$ is a set of edges.*

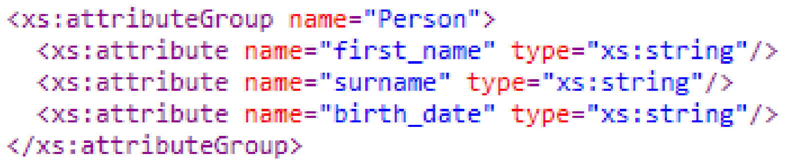
5.1.1 Definition of Equivalence

Before searching for redundancy in SAP we need to have the equivalence of APs defined formally. Since each AP can contain some declarations of attributes, we need to define equivalence of attribute declarations, as well.

Definition 8 *In an arbitrary output of the basic translation algorithm two attribute declarations x, y are equivalent, denoted $x \equiv y$, if their* name, use, default *and* type *attributes have the same values. AP_i, $AP_j \in SAP$ are equivalent, denoted $AP_i \equiv AP_j$, if the following formulas are true:*

$$\forall\ x \in AR_{AP_i}\ \exists\ y \in AR_{AP_j} : x \equiv y$$
$$\forall\ y \in AR_{AP_j}\ \exists\ x \in AR_{AP_i} : y \equiv x$$

Imagine we have an attribute group, like the one in Figure 5.3. Its name is *Person*. The AP, which can be found in *Person*, contains three attribute declarations. To illustrate Definition 8, in Figure 5.4, we show some APs that are equivalent to the *Person*'s one. In Figure 5.5, there are sample APs that are not equivalent to it.



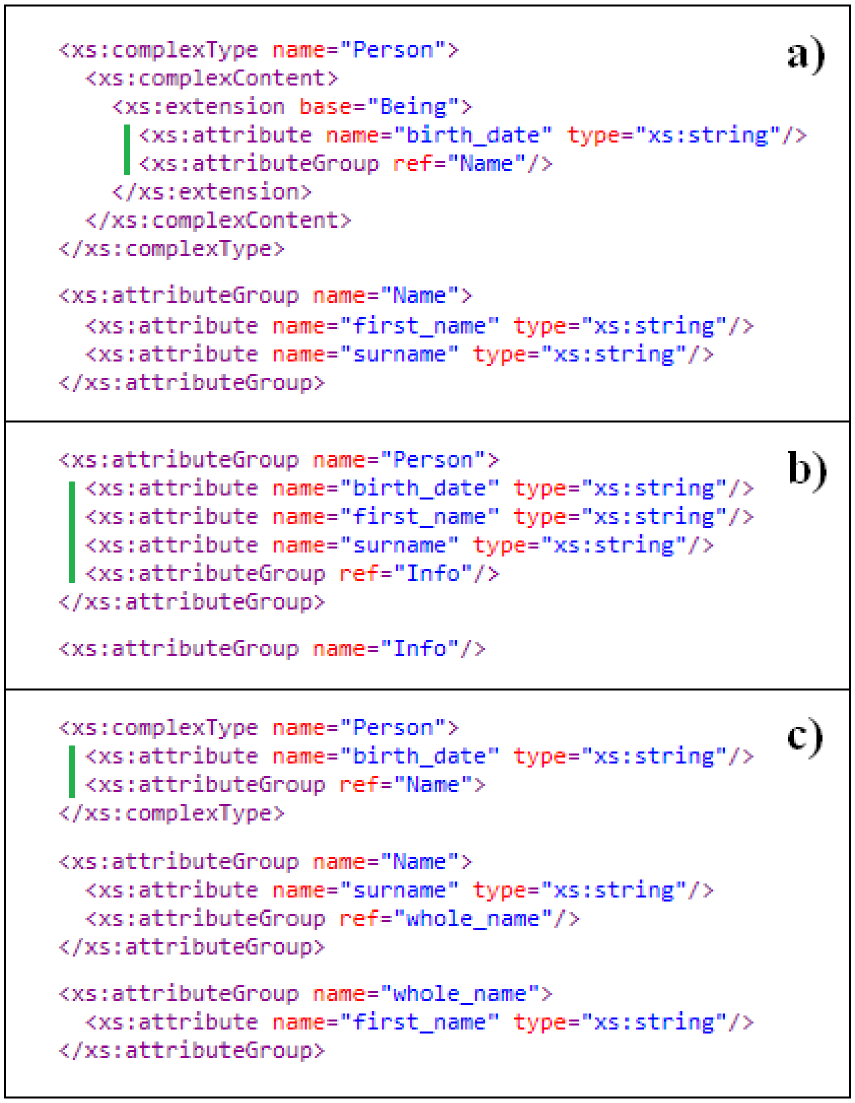
```
<xs:attributeGroup name="Person">
  <xs:attribute name="first_name" type="xs:string"/>
  <xs:attribute name="surname" type="xs:string"/>
  <xs:attribute name="birth_date" type="xs:string"/>
</xs:attributeGroup>
```

Figure 5.3: Sample AP

Let's have a look on the equivalent APs in Figure 5.4. In Figure 5.4 (a), a parent of green AP is an *extension* construct. The AP has one attribute declaration in its AD_{AP} set, the other two attributes are declared in AP hanging under attribute group *Name*, which the green AP depends on.

In Figure 5.4 (b), the green AP contains all the three attribute declarations and a reference to attribute group *Info*. The green AP is equivalent to the one in Figure 5.3 because attribute group *Info* is empty.

Figure 5.4 (c) is very similar to (a), except for the fact that declarations of the three attributes are divided into three APs and the fact that the green



a)

```
<xs:complexType name="Person">
  <xs:complexContent>
    <xs:extension base="Being">
      <xs:attribute name="birth_date" type="xs:string"/>
      <xs:attributeGroup ref="Name"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:attributeGroup name="Name">
  <xs:attribute name="first_name" type="xs:string"/>
  <xs:attribute name="surname" type="xs:string"/>
</xs:attributeGroup>
```

b)

```
<xs:attributeGroup name="Person">
  <xs:attribute name="birth_date" type="xs:string"/>
  <xs:attribute name="first_name" type="xs:string"/>
  <xs:attribute name="surname" type="xs:string"/>
  <xs:attributeGroup ref="Info"/>
</xs:attributeGroup>

<xs:attributeGroup name="Info"/>
```

c)

```
<xs:complexType name="Person">
  <xs:attribute name="birth_date" type="xs:string"/>
  <xs:attributeGroup ref="Name">
</xs:complexType>

<xs:attributeGroup name="Name">
  <xs:attribute name="surname" type="xs:string"/>
  <xs:attributeGroup ref="whole_name"/>
</xs:attributeGroup>

<xs:attributeGroup name="whole_name">
  <xs:attribute name="first_name" type="xs:string"/>
</xs:attributeGroup>
```

Figure 5.4: Equivalent *AP*s

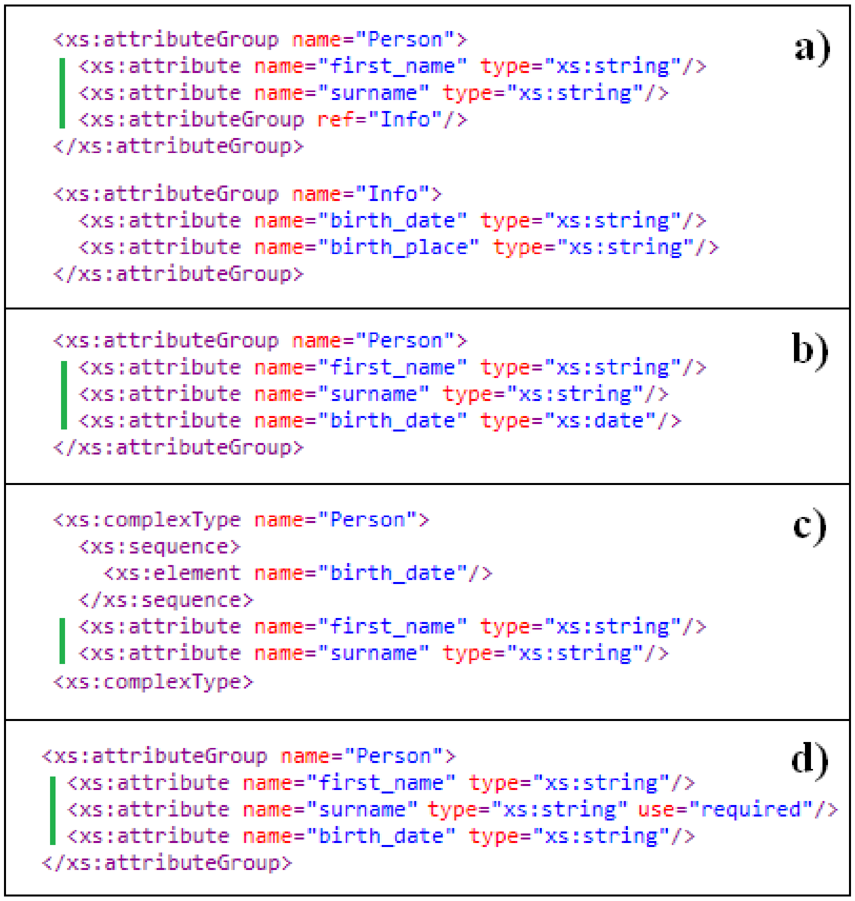

a)
```
<xs:attributeGroup name="Person">
  <xs:attribute name="first_name" type="xs:string"/>
  <xs:attribute name="surname" type="xs:string"/>
  <xs:attributeGroup ref="Info"/>
</xs:attributeGroup>

<xs:attributeGroup name="Info">
  <xs:attribute name="birth_date" type="xs:string"/>
  <xs:attribute name="birth_place" type="xs:string"/>
</xs:attributeGroup>
```
b)
```
<xs:attributeGroup name="Person">
  <xs:attribute name="first_name" type="xs:string"/>
  <xs:attribute name="surname" type="xs:string"/>
  <xs:attribute name="birth_date" type="xs:date"/>
</xs:attributeGroup>
```
c)
```
<xs:complexType name="Person">
  <xs:sequence>
    <xs:element name="birth_date"/>
  </xs:sequence>
  <xs:attribute name="first_name" type="xs:string"/>
  <xs:attribute name="surname" type="xs:string"/>
<xs:complexType>
```
d)
```
<xs:attributeGroup name="Person">
  <xs:attribute name="first_name" type="xs:string"/>
  <xs:attribute name="surname" type="xs:string" use="required"/>
  <xs:attribute name="birth_date" type="xs:string"/>
</xs:attributeGroup>
```

Figure 5.5: Non-Equivalent APs

AP is under a complex type declaration instead of under an *extension* construct.

Note that equivalence of two APs is affected neither by order in which attribute declarations are written, nor by the fact they are declared in another AP which the original AP depends on. The only important aspect is a content of AR_{AP} sets of given APs.

In Figure 5.5 none of four green APs is equivalent to the one in Figure 5.3. In Figure 5.5 (a), AR_{AP} set of a green AP has more items than it should have, because a declaration of an attribute *birth_place* doesn't have its equivalent in an AR_{AP} set of the original AP.

A green AP from Figure 5.5 (b) declares attribute *birth_date* with different data type than expected.

Figure 5.5 (c) contains a green AP, in which the attribute *surname* is declared as required. Again, that's in contrast to the original AP from Figure 5.3.

5.1.2 Redundant Attribute Parts

In this section we introduce an algorithm for removal of redundant APs from an arbitrary output of the basic translation algorithm, i.e. from an instance of xs_{out}.

First of all, we construct a division of all attribute declarations into equivalence classes. According to Definition 8, the fact that an attribute declarations belongs to a given equivalence class can be verified by examining the values of its *name*, *type*, *use* and *default* attributes.

As a second step of the algorithm, SAP is divided into sets of mutually equivalent items. Because the equivalence of APs is given by contents of their AR_{AP} sets (see Definition 8) and AR_{AP} sets are often too large to compare, first, we apply a perfect hash function on AR_{AP} sets and compare only the hash codes.

We work with a perfect hash function on a theoretical level only. In an implementation of a proposed algorithm, we use non-perfect hash function, instead, as explained in Algorithm Analysis at the end of this section.

Two APs are equivalent, if hash codes of their AR_{AP} sets are equal.

When we have a division of SAP into equivalence classes, we try to replace each equivalence class by a single AP, which leads to removal of a redundancy between the APs. This is because no two of them stay mutually equivalent.

A replacement of an equivalence class with a single node is a contraction of GAP according to its subgraph, as said in Definition 9.

Definition 9 *Contraction G.T is a graph that arises from G, if a subgraph T is replaced with a single node t. In addition, node t is connected with all nodes of G, which don't belong to T and which are adjacent to at least one node of the subgraph T.*

When performing a contraction with GAP as a graph, and its equivalence class as a subgraph, the resulting node t is an AP with an attribute group as its parent. We call it a *representative* of the equivalence class.

As a limitation, a contraction of the whole equivalence class is not always possible. It can be done with all equivalent APs that have an attribute groups as their parents. If an AP has a complex type or an *extension* construct as its parent, we don't remove it from GAP. Instead, its content is changed to a reference to the *representative*.

If there is no AP with an attribute group as its parent in an equivalence class, then one such an AP is created and considered to be the *representative*. Again, contents of all other APs from the same equivalence class are replaced with a reference to the *representative*.

If there is an $AP \in V_{GAP}$ directly dependent on any AP from the contracted equivalence class, the edge $E \in E_{GAP}$ going from it is redirected to the *representative*.

In Algorithm 1, the proposed algorithm for elimination of redundant APs is written in pseudocode. For clarity, in the pseudocode we don't describe how a structure of input XML schema document changes. Only, we describe all the modifications of GAP and assume that the structure of the document adapts automatically to a new form of the graph.

For example, when we add a new edge (AP_i, AP_j) into a set of GAP's edges, we assume that a reference on parent of AP_j is added automatically to the content of AP_i. Where it is not clear how exactly a structure of XML schema document is modified, we explain the situation additionally.

At line 2 a function *getSEC()* is called. Its purpose is to find a set of equivalence classes of GAP's vertices. The set is found using a perfect hash function of the AR_{AP}. Each equivalence class is a set of GAP's vertices.

Then we process the equivalence classes one by one. At lines 4-6 we stop

```
Input: $GAP$
Output: $GAP'$

1  $GAP' \leftarrow GAP$
2  $SEC \leftarrow getSEC(GAP')$

3  foreach $EC \in SEC$ do
4      if $|EC| < K$ then
5          goto 3
6      end

7      $AP_{best} \leftarrow best(EC)$
8      $AP_{rep} \leftarrow AP_{best}$

9      if $parent(AP_{best}) \neq$ "$AG$" then
10         $AP_{rep} \leftarrow new(AP_{best})$
11         $V_{GAP'} \leftarrow V_{GAP'} \cup AP_{rep}$
12     end

13     $X \leftarrow \{ AP \in EC \mid parent(AP) =$ "$AG$" $\}$
14     $Y \leftarrow EC \setminus X$

15     if $X \neq \emptyset$ then
16         $V_{-} \leftarrow X \setminus AR_{rep}$
17         $E_{-} \leftarrow \{ (AP_i, AP_j) \mid AP_i \in V_{-} \vee AP_j \in V_{-} \}$
18         $E_{+} \leftarrow \{ (AP_i, AP_{rep}) \mid \exists AP_j \in V_{-} \colon (AP_i, AP_j) \in E_{GAP'} \}$

19         $V_{GAP'} \leftarrow V_{GAP'} \setminus V_{-}$
20         $E_{GAP'} \leftarrow E_{GAP'} \setminus E_{-} \cup E_{+}$
21     end

22     foreach $AP \in Y$ do
23         $clear(AP)$
24         $E_{GAP'} \leftarrow E_{GAP'} \cup (AP, AP_{rep})$
25     end
26 end
```

Algorithm 1: Removal of Redundant APs

processing of equivalence classes that have less than K items. K is a number determined by a value of a special parameter from a user configuration.

$$(\; min\_apec\_size \; , \; v \;) \in C$$

$$v \in \{ \; 2 \ldots * \; \}$$

At line 10 there is a function *best()*. It gives an AP, the content of which is the best one for representing the equivalence class. The word *best* may be interpreted in many ways. For example, the function may return such an AP, which has the smallest cardinality of AD_{AP}.

However, the function mustn't give an AP, which depends on another AP from the same equivalence class. That's because we want the result of *best()* to become a *representative* of the equivalence class after a contraction.

If the *best* AP doesn't have an attribute group as its parent, it cannot be used as a *representative* of EC, because it cannot be referenced, but its content is still the *best* one.

In that case, we create a new AP and a new attribute group as its parent. Then, we copy the content of AP_{best} into the new one. We add the new AP into the graph and assign it to AP_{rep}. See lines 9-12.

AP_{rep} is a *representative* of the processed EC.

At line 13 we assign a set of all APs with attribute groups as their parents to X. Function *parent()* returns a string 'AG' if a parent of a given parameter is an attribute group. It returns 'EX' if it is an *extension* contruct and 'CT' if a parent is an complex type declaration. In contrast, Y is a set of all APs that doesn't have attribute group as their parents.

A contraction of a graph is performed at lines 19 and 20. It uses the sets, prepared previously at lines 16-18. When removing an edge from GAP' the algorithm should always verify that the edge is not the only one pointing to the target AP.

If it is and the target AP has an attribute group as its parent, the whole AP must be removed from GAP', as well as the attribute group must be removed from a structure of XML schema document.

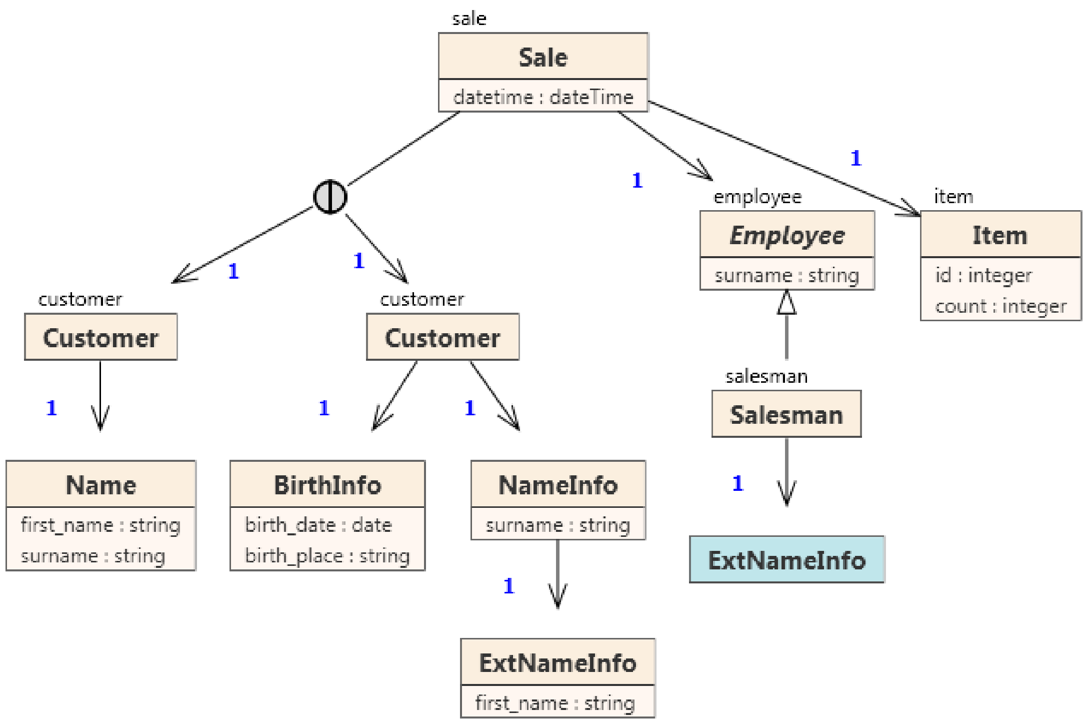


Figure 5.6: XSEM-H View, Redundant *AP*s Elimination Input

At lines 22-25, first, a content of each $AP \in Y$ is removed. Then, a new edge pointing from AP to the *representative* is added into GAP', so AP gets a reference to the parent attribute group of AP_{rep}.

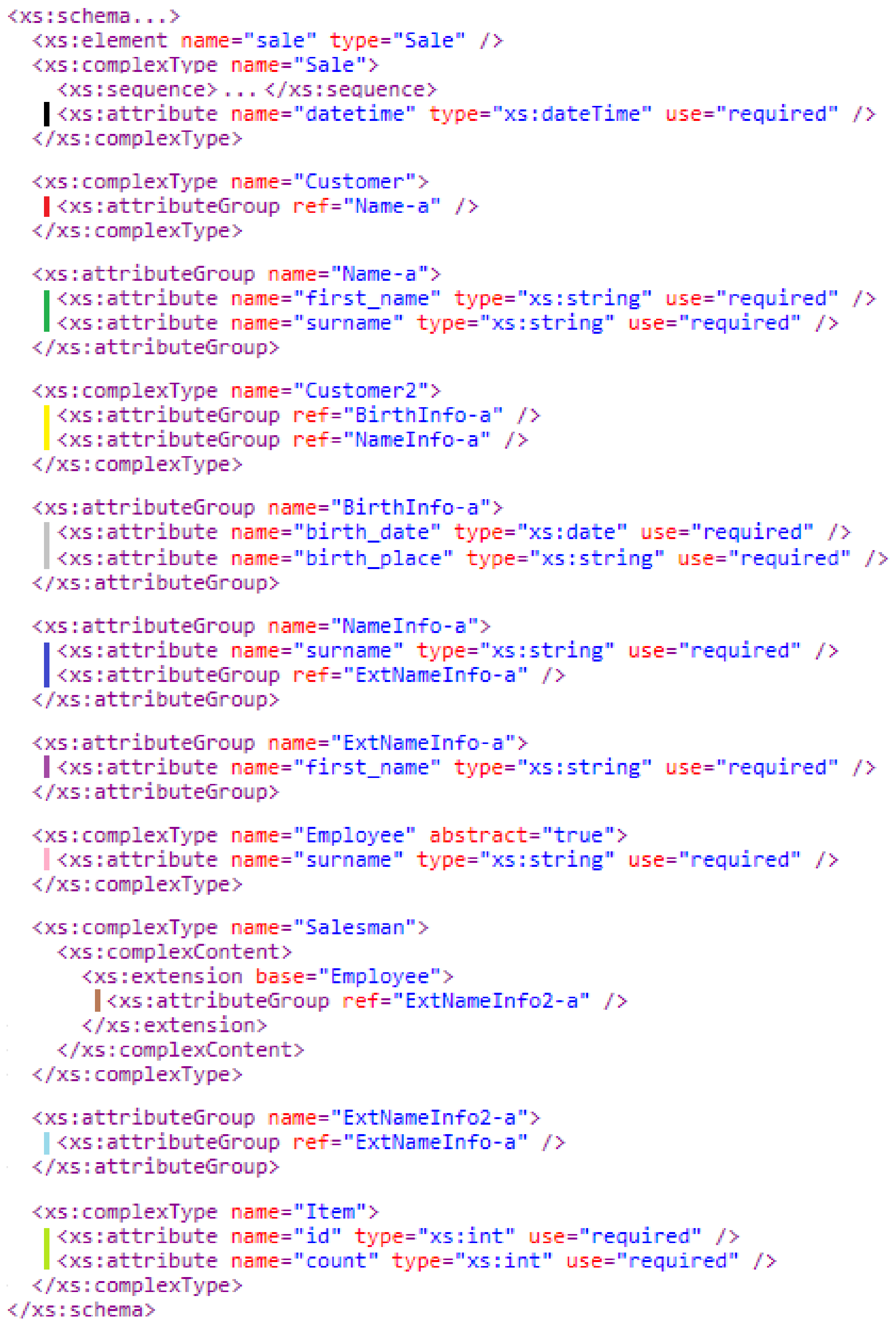
We illustrate Algorithm 1 on an example. Figure 5.6 shows an XSEM-H view representing structure of XML documents, used for evidence of sales. For each sale we store information about a time when it was made, on salesman, on items sold and on customer, of course. The XSEM-H view could be simpler, but our objective here is to produce as much attribute redundancy as possible on relatively small sample.

In Figure 5.7 there is an XML schema, expressed in XML Schema language, resulting from translation of the XSEM-H view. Eleven APs can be found in it and again we identify them by color.

For now, let us suppose that attribute declarations are represented by values of their *name* attributes. In this example, this is possible because no two attribute declarations with the same values of *name* attributes differ in values of *type*, *use* and *default* attributes. Then obviously,

$$
\begin{aligned}
&AR_{AP_{black}} = \{datetime\}\\
&AR_{AP_{red}} = \{first\_name, surname\}\\
&AR_{AP_{dgreen}} = \{first\_name, surname\}\\
&AR_{AP_{yellow}} = \{birth\_date, birth\_place, surname, first\_name\}\\
&AR_{AP_{grey}} = \{birth\_date, birth\_place\}\\
&AR_{AP_{dblue}} = \{surname, first\_name\}\\
&AR_{AP_{violet}} = \{first\_name\}\\
&AR_{AP_{pink}} = \{surname\}\\
&AR_{AP_{brown}} = \{first\_name\}\\
&AR_{AP_{lblue}} = \{first\_name\}\\
&AR_{AP_{lgreen}} = \{id, count\}
\end{aligned}
$$

From Definition 8 it's clear that dark green, red and dark blue APs are equivalent. At the same time, light blue and violet APs are equivalent.



```
<xs:schema...>
  <xs:element name="sale" type="Sale" />
  <xs:complexType name="Sale">
    <xs:sequence>...</xs:sequence>
    <xs:attribute name="datetime" type="xs:dateTime" use="required" />
  </xs:complexType>

  <xs:complexType name="Customer">
    <xs:attributeGroup ref="Name-a" />
  </xs:complexType>

  <xs:attributeGroup name="Name-a">
    <xs:attribute name="first_name" type="xs:string" use="required" />
    <xs:attribute name="surname" type="xs:string" use="required" />
  </xs:attributeGroup>

  <xs:complexType name="Customer2">
    <xs:attributeGroup ref="BirthInfo-a" />
    <xs:attributeGroup ref="NameInfo-a" />
  </xs:complexType>

  <xs:attributeGroup name="BirthInfo-a">
    <xs:attribute name="birth_date" type="xs:date" use="required" />
    <xs:attribute name="birth_place" type="xs:string" use="required" />
  </xs:attributeGroup>

  <xs:attributeGroup name="NameInfo-a">
    <xs:attribute name="surname" type="xs:string" use="required" />
    <xs:attributeGroup ref="ExtNameInfo-a" />
  </xs:attributeGroup>

  <xs:attributeGroup name="ExtNameInfo-a">
    <xs:attribute name="first_name" type="xs:string" use="required" />
  </xs:attributeGroup>

  <xs:complexType name="Employee" abstract="true">
    <xs:attribute name="surname" type="xs:string" use="required" />
  </xs:complexType>

  <xs:complexType name="Salesman">
    <xs:complexContent>
      <xs:extension base="Employee">
        <xs:attributeGroup ref="ExtNameInfo2-a" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:attributeGroup name="ExtNameInfo2-a">
    <xs:attributeGroup ref="ExtNameInfo-a" />
  </xs:attributeGroup>

  <xs:complexType name="Item">
    <xs:attribute name="id" type="xs:int" use="required" />
    <xs:attribute name="count" type="xs:int" use="required" />
  </xs:complexType>
</xs:schema>
```

Figure 5.7: XML Schema, Redundant *AP*s Elimination Input

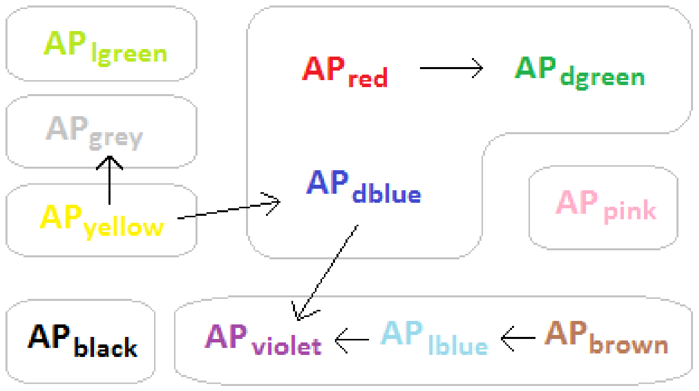


Figure 5.8: Before Redundant APs Elimination

Therefore, seven equivalence classes of APs exist in this example. We illustrate them in Figure 5.8 with grey borders. Also, we show all existing direct dependencies between APs.

The algorithm processes equivalence classes (grey rectangles), one by one. Let us suppose that in a user configuration the parameter with key *min_apec_size* is set to 2. Then, processing of rectangles with fewer than 2 items ends at line 3 of Algorithm 1, because the condition is not satisfied.

When processing a rectangle with light blue, brown and violet APs, obviously, the light blue AP and brown AP cannot be *representatives*, because they are directly dependent on an AP from the same equivalence class. Therefore, AP_{violet} is assigned to AP_{rep}.

The light blue AP and the violet AP are assigned to X at line 13. During a contraction at lines 19-20, the AP_{lblue} is completely removed from GAP' and one new edge is added. The new edge leads from the brown AP to the violet one.

When processing the rectangle with three APs, function *best()* gives dark blue AP. Therefore, the dark blue AP is called AP_{rep}. The dark blue and dark green are assigned to X, because their parents are attribute group declarations. The third AP (the red one) is assigned to variable Y.

During contraction of X dark green AP is removed from the graph and the edge leading to it is redirected to AP_{rep}. Finally, a content of red AP is replaced by reference to a parent attribute group of AP_{rep}.

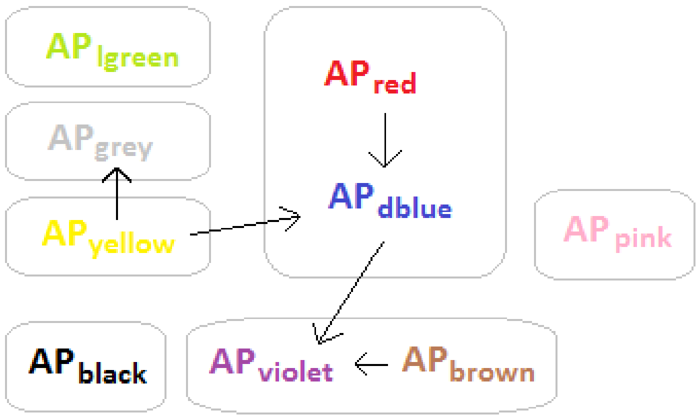


Figure 5.9: After Redundant *AP*s Elimination

Figure 5.9 illustrates how the Figure 5.8 changes after running the algorithm for elimination of redundant *AP*s. Figure 5.10 shows the result of basic translation after running the algorithm (compare with Figure 5.7).

Obviously, the algorithm for elimination of redundancies in *AP*s doesn't remove all redundancies that can occur in attribute declarations.

For example, an attribute declaration with name *surname*, type *xs:string* and use set to *required* is included both in pink and dark blue *AP*. Ideally, the attribute would be declared only once and its declaration would be referenced from the pink and dark blue *AP*s.

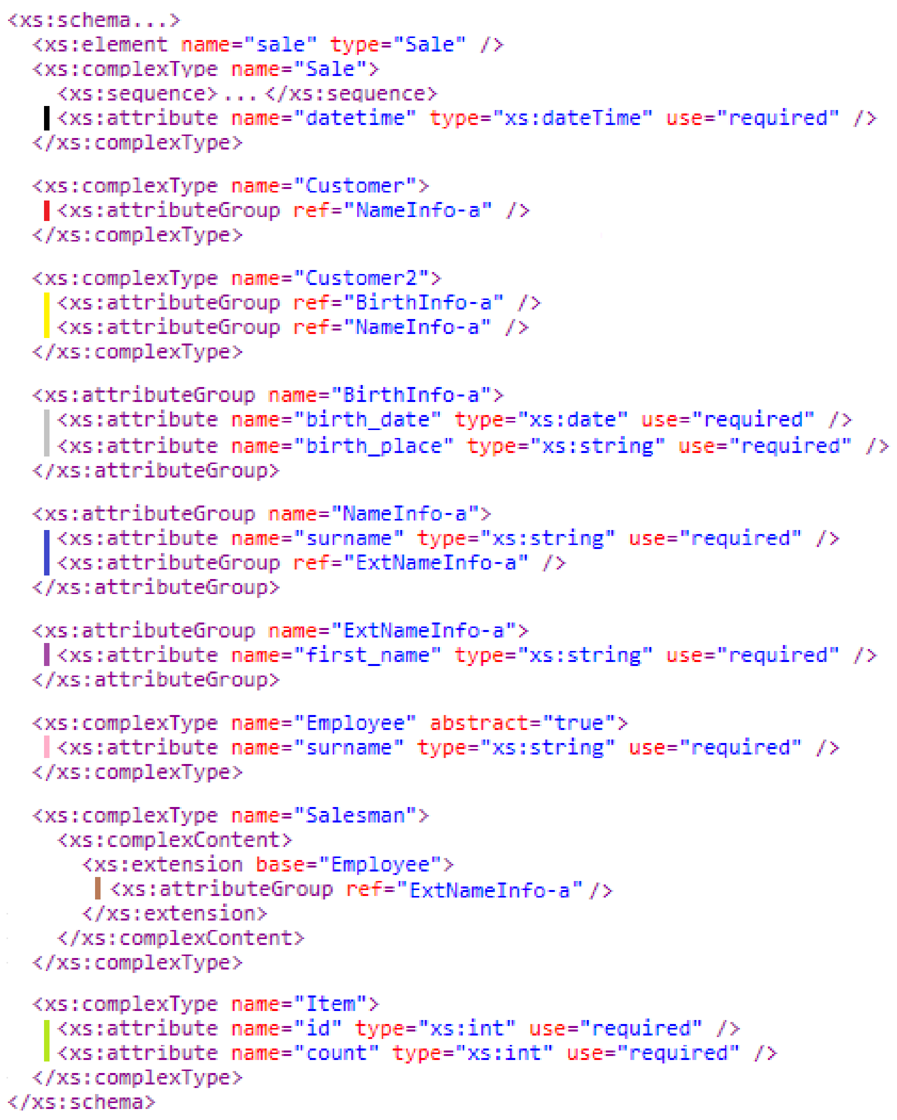
A solution of this problem is proposed in Section 5.1.3.

Algorithm Analysis

First, we show that a **termination** is ensured for every possible input.

In each instance of xs_{OUT} there is a finite number of *AP*s and attribute declarations. For each *AP* we construct a finite set of some attribute declarations, called AR_{AP}. Based on the AR_{AP}, we divide *AP*s into finite number of equivalence classes.

We process the equivalence classes one by one. In each step we can remove some *AP*s from the graph and/or add exactly one *AP* into it. Always, the new



```xml
<xs:schema...>
  <xs:element name="sale" type="Sale" />
  <xs:complexType name="Sale">
    <xs:sequence>...</xs:sequence>
    <xs:attribute name="datetime" type="xs:dateTime" use="required" />
  </xs:complexType>

  <xs:complexType name="Customer">
    <xs:attributeGroup ref="NameInfo-a" />
  </xs:complexType>

  <xs:complexType name="Customer2">
    <xs:attributeGroup ref="BirthInfo-a" />
    <xs:attributeGroup ref="NameInfo-a" />
  </xs:complexType>

  <xs:attributeGroup name="BirthInfo-a">
    <xs:attribute name="birth_date" type="xs:date" use="required" />
    <xs:attribute name="birth_place" type="xs:string" use="required" />
  </xs:attributeGroup>

  <xs:attributeGroup name="NameInfo-a">
    <xs:attribute name="surname" type="xs:string" use="required" />
    <xs:attributeGroup ref="ExtNameInfo-a" />
  </xs:attributeGroup>

  <xs:attributeGroup name="ExtNameInfo-a">
    <xs:attribute name="first_name" type="xs:string" use="required" />
  </xs:attributeGroup>

  <xs:complexType name="Employee" abstract="true">
    <xs:attribute name="surname" type="xs:string" use="required" />
  </xs:complexType>

  <xs:complexType name="Salesman">
    <xs:complexContent>
      <xs:extension base="Employee">
        <xs:attributeGroup ref="ExtNameInfo-a"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="Item">
    <xs:attribute name="id" type="xs:int" use="required" />
    <xs:attribute name="count" type="xs:int" use="required" />
  </xs:complexType>
</xs:schema>
```

Figure 5.10: XML Schema, Redundant APs Elimination Output

AP belongs to the equivalence class, that is currently processed. That means, the number of equivalence classes cannot be changed by processing any one of them.

A **correctness** of the algorithm comes from the following facts.

After each class of equivalence is processed, it contains exactly one AP with attribute group as its parent. We call it a *representative*. In other words, no redundant AP with such a parent left in the XML schema.

All the other APs from EC are still mutually equivalent, but content of each of them is a simple reference to the *representative*. Therefore, such APs are not considered redundant.

The algorithm is correct because the processing of one specific equivalence class cannot make an AP from an equivalence class, which has been already processed, redundant.

For a **time analysis**, let us suppose that we have a DOM [19] structure of XML Schema document, as a result of the translation algorithm. Also, let us suppose that m denotes a number of APs in DOM and n is a number of attribute declarations in DOM.

First, the algorithm looks for a division of APs into equivalence classes. For each AP it must construct its AR_{AP}. It must collect the attribute declarations from the AP and all the APs, it depends on. In the worst case, a construction of the AR_{AP}s of every APs would require $m * n$ steps.

Fortunatelly, we don't need to go through the whole document especially for this purpose. We can create a list of APs and their AR_{AP}s directly during the translation of an XSEM-H view into the DOM structure.

A structure representing AR_{AP} is organized as a binary heap. Items of AR_{AP} are not attribute declarations, but equivalence classes of attribute declarations. Again, an equivalence class of each attribute declaration is found during the creation of the DOM structure.

For each AR_{AP} structure, we call *min* and *deletemin* methods iteratively and concatenate returned values into a string. This require $m * log(n)$ steps. The string is then used as an input of a hash function.

In the formal description of the algorithm, we assumed a perfect hash function. In fact, we uses MD5, SHA-1 [20] or any other well-known hash function, instead. A result of the hash function represents an equivalence class for the processed AP. Because used hash functions are not perfect, always, we must made an additionally comparison of AR_{AP}s to be sure, that AP really belongs to the proposed equivalence class.

When we have a division of SAP into the classes of equivalence, we process these classes one by one. For each equivalence class, we simply need to go through its items to perform all the required modifications.

5.1.3 Redundant Attribute Declarations

In this section we introduce an algorithm for removal of redundant attribute declarations from an arbitrary output of the basic translation.

We propose an algorithm that operates with AD_{AP} sets (see Definition 6) of all APs in an XML schema. Its purpose is to make those AD_{AP} mutually disjoint, so that no two AP_i, $AP_j \in SAP$ contain the same attribute declaration.

Definition 10 *SAP is without redundant attribute declarations iff for every AP_i, $AP_j \in SAP$ where $AP_i \neq AP_j$, sets of attribute declarations contained in them directly are disjoint, i.e. $AD_{AP_i} \cap AD_{AP_j} = \emptyset$.*

The algorithm runs in steps until all redundancies from attribute declarations are removed. In each step, it gets a subset of SAP and calculates an intersection of AD_{AP} of all the subset's APs.

Then, attribute declarations from the intersection are excluded into a new separate attribute group declaration. In place of the attribute declarations, each participating AP gets an reference to the new attribute group.

Thus, the algorithm adds new nodes into GAP and changes a set of its edges, repeatedly.

We must define a rule for selecting SAPs subsets in each step of the algorithm. Note that an output of the algorithm can vary for different rules applied. In this algorithm, as the subset of SAP, we always select two APs that have the largest intersections of AD_{AP} sets.

In other words, we select $AP_i, AP_j \in SAP$ that satisfy Equation 5.2. If two and more pairs of APs satisfy it, we can select any pair of them.

$$\neg\, \exists AP_m, AP_n \in SAP\colon\ |AD_{AP_m} \cap AD_{AP_n}\ | > |AD_{AP_i} \cap AD_{AP_j}\ | \tag{5.2}$$

In Algorithm 2, the proposed algorithm is written in pseudocode. Let us describe its most important parts.

At lines 2-12, for each pair of APs an information on their intersection is added into a structure called S. Each record of an intersection consists of a set of shared attribute declarations (X) and indexes of both APs that share it.

The algorithm assumes that S keeps records ordered by size of X, in descending order. S should be some kind of a priority queue, e.g. binary heap. If S is not organized as a priority queue, then at line 14, repeated selection of items with the biggest X is very inefficient.

At lines 13-26, items are extracted from S successively. In each step, new AP is created from the intersection X (line 15) and inserted into GAP', as seen at lines 22-23. Contents of both the APs sharing the X are changed, according to the formulas proposed at lines 24-25.

Now, it seems that everything is finished. But that's not true at all. The algorithm have passed through all the intersections, found at the very beginning while exploring $V_{GAP'} = SAP$. The problem is that $V_{GAP'}$ changes during a process of the algorithm and therefore, a recalculation of S is neccessary after each step of a *while* cycle (13-26). Well, let us propose a solution to the problem.

```
Input: $GAP$
Output: $GAP'$

1  $GAP' \leftarrow GAP$

2  foreach $AP_i \in V_{GAP'}$ do
3      foreach $AP_j \in V_{GAP'}$ do
4          if $j \leq i$ then
5              goto 3
6          end
7          $X \leftarrow AD_{AP_i} \cap AD_{AP_j}$

8          if $X \neq \emptyset$ then
9              $S.add(X,\, i,\, j)$
10         end
11     end
12 end

13 while $S \neq \emptyset$ do
14     $(X,\, i,\, j) \leftarrow S.getmax()$
15     $AP_{new} \leftarrow new(X)$

16     foreach $AP_k \in V_{GAP'}$ do
17         $Y \leftarrow AD_{AP_i} \cap AD_{AP_j}$

18         if $Y \neq \emptyset$ then
19             $S.add(Y,\, k,\, new)$
20         end
21     end

22     $V_{GAP'} \leftarrow V_{GAP'} \cup AP_{new}$
23     $E_{GAP'} \leftarrow E_{GAP'} \cup (AP_i, AP_{new}) \cup (AP_j, AP_{new})$

24     $AP_i \leftarrow AP_i \setminus X \cup ref(AP_{new})$
25     $AP_j \leftarrow AP_j \setminus X \cup ref(AP_{new})$
26 end
```

Algorithm 2: Removal of Redundant Attribute Declarations

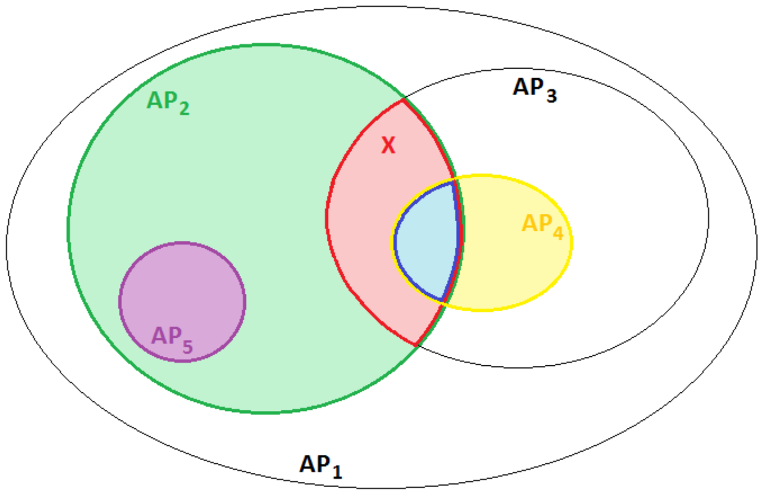


Figure 5.11: Relationships of Intersections

Ensuring Consistency

At the very beginning of the algorithm, we create and fill structure S, so that it represents intersections of AD_{AP} sets of each pair from $V_{GAP'}$. Then, we get items of S one-by-one and in each step we change some vertices and edges of GAP'.

The problem is that such an approach isn't correct, because after each change of GAP' the set S is no more consistent and its recalculation is neccessary.

Because the recalculation of S in each step of the algorithm would be very time-consuming approach, here, we propose an easier way. We define rules determining how to change S after a change of GAP', to ensure data consistency.

The rules must be applied in the *getmax()* function at line 14. Moreover, because in each step of the *while* cycle we create one new AP, we must add some new intersections into $V_{GAP'}$, as seen at line 19.

Figure 5.11 illustrates a space of attribute declarations and attribute group references of an arbitrary output of the basic translation algorithm. Five APs are in the space. They are identified by numbers and Figure 5.11 clearly illustrates their intersections. Obviously

$V_{GAP} = SAP = \{AP_1, AP_2, AP_3, AP_4, AP_5\}$

Let us suppose that at lines 13-26 of Algorithm 2 we work with the red intersection of AD_{AP_2} and AD_{AP_3} called X, i.e. $i = 2$ and $j = 3$.

We want to create a new AP from X and exclude X from both the APs. We are interested in how intersections of pairs of the other APs can be affected with those actions, i.e. how S can be damaged.

There are six types of intersections according to their relationships to the intersection X. The first type is intersection, the participants of which are completely different then participants of X. The exclusion of X from S doesn't affect intersections of the first type at all.

The five other types are listed in Table 5.1. Both in Figure 5.11 and the table, the types are distinguished by color.

In Table 5.1, there are characteristic formulas for the types. In each formula, Y denotes an instance of the appropriate type. In column *Figure 5.11 Example* there is a sample value of Y, taken from Figure 5.11.

Table 5.1: Types of Intersections according to their Relations to X

Type	Characteristic Formula	Figure 5.11 Example
T_{green}	$\forall\, Y \in T_{green}\colon Y \supset X$	$AD_{AP_1} \cap AD_{AP_2}$
T_{red}	$\forall\, Y \in T_{green}\colon Y = X$	$AD_{AP_2} \cap AD_{AP_3}$
T_{blue}	$\forall\, Y \in T_{green}\colon Y \subset X$	$AD_{AP_2} \cap AD_{AP_4}$
T_{violet}	$\forall\, Y \in T_{green}\colon Y \cap X = \emptyset$	$AD_{AP_2} \cap AD_{AP_5}$
T_{yellow} .	$\forall\, Y \in T_{green}\colon Y \cap X \neq \emptyset \wedge X \setminus Y \neq \emptyset \wedge Y \setminus X \neq \emptyset$	$AD_{AP_3} \cap AD_{AP_4}$.

A green type T_{green} includes those intersections from S that represent a superset of X. In fact, no intersection of green type can exist, because the rule for selection of X (see Equation 5.2) says that X is always the biggest intersection. Thus, from now, we don't consider green intersections at all.

T_{blue} (a blue type) contains intersections that represent a subset of X.

If an intersection of AD_{AP} sets of two APs shares some items with X and doesn't belong to T_{blue} nor T_{green}, then it belongs to a yellow type T_{yellow}.

T_{violet} represents all intersections that don't intersect with X at all.

A red type is for intersections that have the same items like X.

An exclusion of X from AP_2 and AP_3 into a new AP, realized in one step of the Algorithm 2, doesn't affect intersections from T_{violet}. Therefore, in *get-max()* function the intersections from T_{violet} in S are not modified.

A content of intersections of the blue type is not changed at all, but one participant of each intersection must be changed to the new AP.

Let us illustrate the situation in Figure 5.11. When new AP is created from X, then AD_{AP_2} and AD_{AP_3} sets are reduced to $AD_{AP_2} \setminus X$ and $AD_{AP_3} \setminus X$. So the blue type intersection can no longer be equal to $AD_{AP_2} \cap AD_{AP_4}$. It must be modified to $AD_{AP_{NEW}} \cap AD_{AP_4}$.

In fact, we don't need to do such a modification, because in each step, the Algorithm 2 automatically adds intersections of the new AP into S. Thus, because $AD_{AP_{NEW}} \cap AD_{AP_4}$ is added into S at line 19, each $Y \in T_{blue}$ can be simply removed from S.

An intersection $Y \in T_{yellow}$ must be modified to $Y \setminus AD_{AP_{NEW}}$. Moreover, one new intersection must be added into S, but that operation has nothing to do with ensuring consistency of S. The operation is performed at line 19.

We illustrate Algorithm 2 on an example. In Figure 5.12 a) there are four APs and some equivalence classes of attribute declarations. The equivalence classes are represented by letters of the alphabet. If an AP_i contains a letter L, the attribute declaration represented by equivalence class of L belongs to AD_{AP_i}.

In the first phase of the algorithm we fill the structure S. Records of S are always kept ordereb by size of set X.

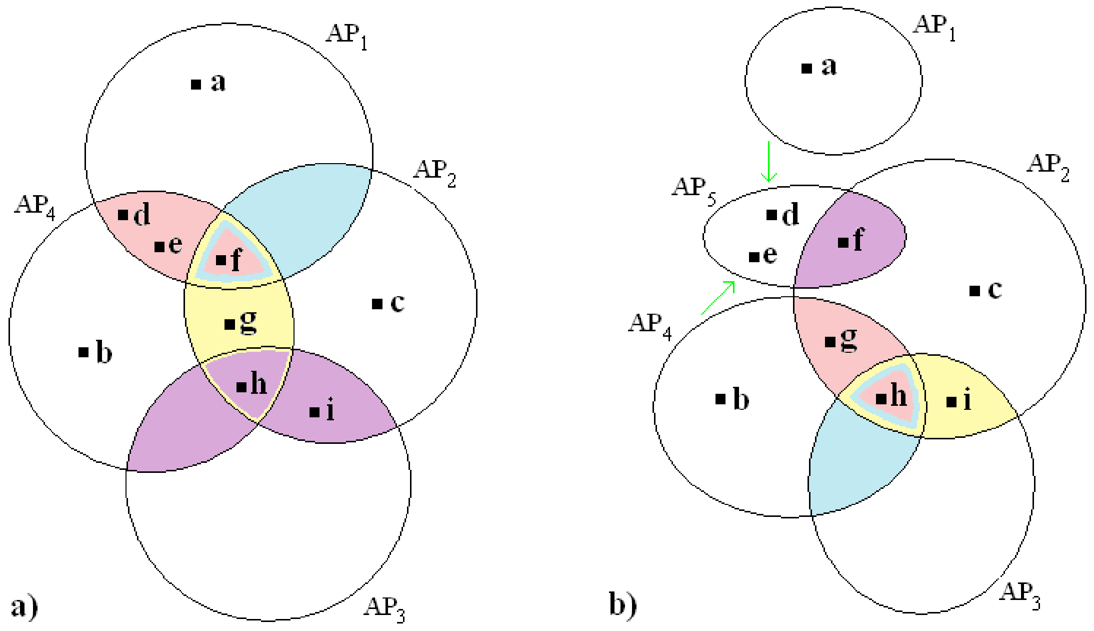


Figure 5.12: Redundant Attribute Declarations Elimination, Part 1

$$S_0 = \{\ (\{d,e,f\},1,4),(\{f,g,h\},2,4),(\{h,i\},2,3),(\{f\},1,2),(\{h\},3,4)\ \}$$

In its second phase the algorithm runs in steps, as said already. In each step the function *getmax()* gives the first item of S and perform some modifications of its other items to ensure consistency of S.

Well, *getmax()* gives $(\{d,e,f\},1,4)$. In Figure 5.12 a) we can see which item of S belongs to which type according to their relationship to $X = \{d,e,f\}$.

Following the rules, we remove the blue intersection from S and reduce the yellow intersection from $(\{f,g,h\},2,4)$ to $(\{g,h\},2,4)$,.

And of course, no intersection of violet type is modified.

The next steps of the algorithm are analogous. Figure 5.13 illustrates how the situation changes, and here, we propose the values of structure S for each step of the algorithm.

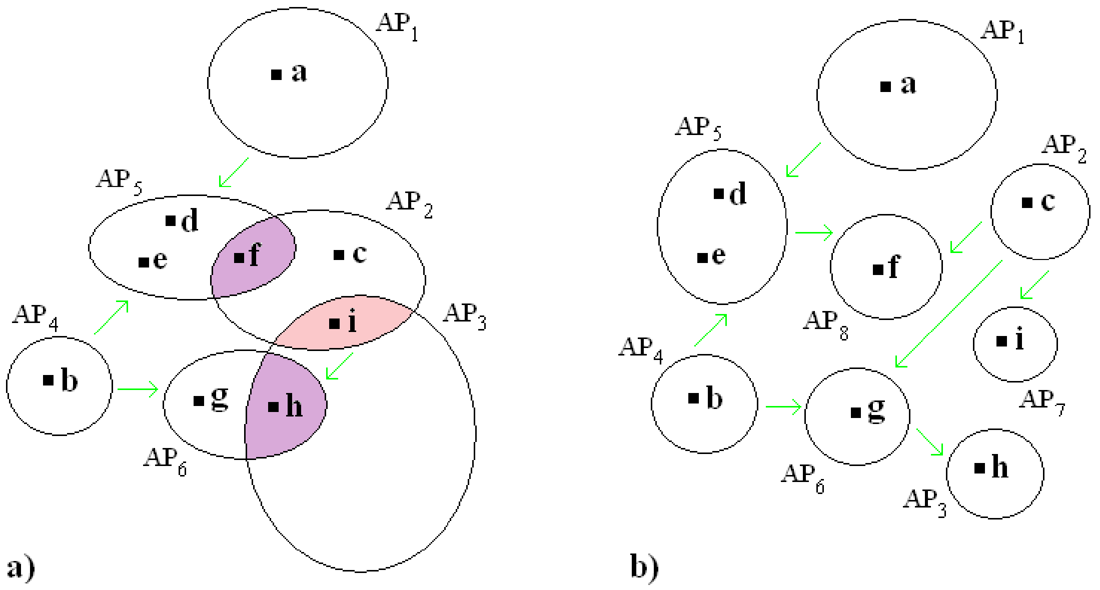


Figure 5.13: Redundant Attribute Declarations Elimination, Part 2

$S_1 = \{\ (\{g, h\}, 2, 4),\ (\{h, i\}, 2, 3),\ (\{f\}, 2, 5),\ (\{h\}, 3, 4)\ \}$
$S_2 = \{\ (\{i\}, 2, 3),\ (\{h\}, 3, 6),\ (\{f\}, 2, 5)\ \}$
$S_3 = \{\ (\{h\}, 3, 6),\ (\{f\}, 2, 5)\ \}$
$S_4 = \{\ (\{f\}, 2, 5)\ \}$

Algorithm Analysis

The algorithm **terminates** for every possible input. That's because in its first phase we examine pairs of APs and obviously, the number of APs is finite. In the second phase, we get items of structure S one by one. We finish, when S is empty. The fact that in each step we add new items into S complicates the situation a little.

However, a termination of the second phase of the algorithm is ensured by the fact, that in each step for at least one equivalence class of attribute declarations the number of AD_{AP}s, in which it is contained, decreases.

A **correctness** of the algorithm is guaranteed by an existence of the rules for ensuring consistency in structure S. Thanks to the rules, after each step S is modified to a state, to which it can be set by a re-start of the whole first phase of the algorithm.

For a **time analysis** let us suppose that m denotes a number of APs in a DOM structure and n is a number of attribute declarations in DOM.

The first phase of the algorithm requires m^2 comparisons of two AD_{AP} sets. Theoretically, an AD_{AP} set can contain all the attribute declarations in the XML schema. Therefore, the whole first phase requires $m^2 * n$ comparisons of attribute declarations (resp. their equivalence classes).

In the second phase, the most important thing is, that the algorithm for ensuring consistency of S is linear.

5.2 Redundancy in Element Declarations

Another type of redundancy in an output of the basic translation algorithm is a redundancy in element declarations. In this section, we describe where exactly such declarations can be found and how do they look like. We explain what can be considered redundant in element declarations and how we can deal with such a redundancy.

However, in this part of the work, we don't create a formal model like in Section 5.1. Instead, we explain how the problem, or at least a part of it, can be converted to the one in Section 5.1.

As seen in Figures 3.5 - 3.8, in an instance of xs_{OUT}, a declaration of an XML element can occur on a global level (under the *xs:schema* element), as well as under a *sequence* or a *choice* constructions.

An XML element is always declared by an *xs:element* element. The declaration contains a *name* attribute, specifying a name of the declared XML element, and a *type* attribute that specifies its type.

If the *type* attribute is not presented, then type of the declared element is given by a complex type, which is declared directly under the element declaration.

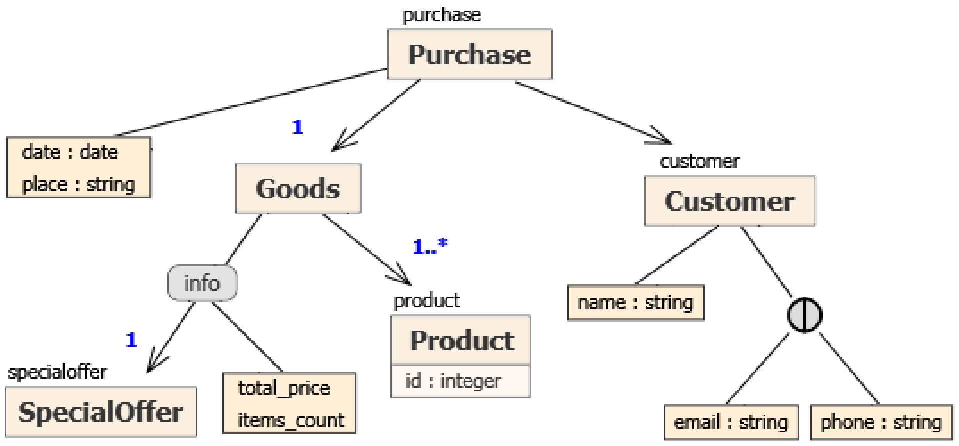


Figure 5.14: XSEM-H View, Redundancy in Element Declarations

When the XML element is declared under a *choice* or a *sequence* construction, moreover, it can contain attributes specifying the minimal and maximal allowed number of occurences of the declared XML element.

Figure 5.14 shows a sample XSEM-H view and in Figure 5.15, we can find a translation of the view into the XML Schema language.

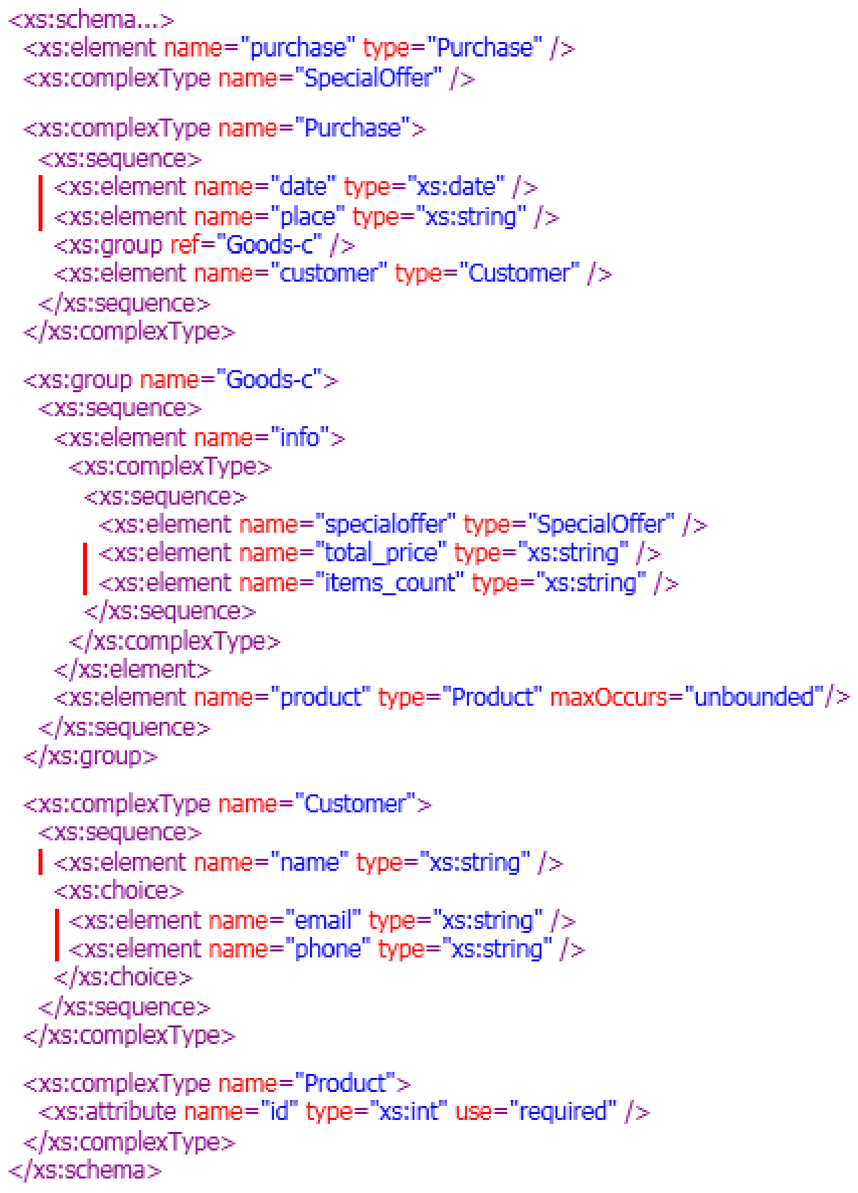
We distinguish two types of element declarations, both the types are illustrated in Figure 5.15. The types are identified by a value of *type* attributes. If the value is a primitive datatype of the XML Schema language [21] (*xs:string*, *xs:date*,...), then the element declaration is of the first type.

Otherwise, the value of the *type* attribute refers to an existing complex type or the *type* is not presented at all. In such cases, the element declarations is considered to be of the second type.

Each element declaration of the first type is created by a translation of an attribute container's item. In other words, such an element declaration is derived from an attribute, which is declared on a conceptual level.

If we declare the same attribute directly in a node of an XSEM-H view, not in the attribute container, then, it is translated into an XML Schema attribute declaration and its redundancy (if any) is solved in Section 5.1.

Therefore, we can solve a redundancy of an element declaration of the first



```
<xs:schema...>
  <xs:element name="purchase" type="Purchase" />
  <xs:complexType name="SpecialOffer" />

  <xs:complexType name="Purchase">
    <xs:sequence>
      <xs:element name="date" type="xs:date" />
      <xs:element name="place" type="xs:string" />
      <xs:group ref="Goods-c" />
      <xs:element name="customer" type="Customer" />
    </xs:sequence>
  </xs:complexType>

  <xs:group name="Goods-c">
    <xs:sequence>
      <xs:element name="info">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="specialoffer" type="SpecialOffer" />
            <xs:element name="total_price" type="xs:string" />
            <xs:element name="items_count" type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="product" type="Product" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:group>

  <xs:complexType name="Customer">
    <xs:sequence>
      <xs:element name="name" type="xs:string" />
      <xs:choice>
        <xs:element name="email" type="xs:string" />
        <xs:element name="phone" type="xs:string" />
      </xs:choice>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Product">
    <xs:attribute name="id" type="xs:int" use="required" />
  </xs:complexType>
</xs:schema>
```

Figure 5.15: XML Schema, Redundancy in Element Declarations

type using methods very similar to the ones, described in Section 5.1.

Finding a redundancy in element declarations of the second type is much more complicated task. We don't deal with the task here, it is out of the scope of this work. To illustrate the problem we use Figure 5.14.

Obviously, each attribute container in the XSEM-H view is a leaf. This is true for all existing XSEM-H views. In other words, the element declarations of the first type cannot be created from inner nodes of the XSEM-H view. Therefore, the element declarations created from the inner nodes are always of the second type.

To check an equivalency of two inner nodes (or element declarations created from them) we must compare their subtrees properly (or we must compare the XML Schema objects that were created by translation of the subtrees). That's not easy at all and is out of the scope of this work.

In Section 5.1 we define a term AP. We can create a similar definition for element declarations of the first type, but we must take into account at least two specifics of element declarations.

First, we must work with sequences of element declaration, not with sets like in the case of attribute declarations. Second, we must consider *minOccurs* and *maxOccurs* attributes.

Definition 11 *Element part, denoted EP, is a sequence of adjacent declarations of elements with primitive types and without* minOccurs *and* maxOccurs *attributes, that have the same parent element in an XML structure of a basic translation output such that*

$$\neg\, \exists EP'\colon EP' \supset EP$$

Figure 5.15 illustrates Definition 11. Each red line on the left side of the XML schema represents one EC.

With Definition 11, we are able to modify the algorithm from Section 5.1.3 to be applicable on the problem of redundancy in element declarations of the first type.

The modified algorithm must take into account the order of element declarations in EC. The intersections of ECs are excluded into model groups, in contrast to the instersections of attribute declarations, that are excluded into attribute groups in the original algorithm.

5.3 Nestings of Choices

The XML schema xs_{OUT} defines a *choice* construction, as illustrated in Figures 3.5 - 3.8. In this section we analyze where in an arbitrary output of the basic translation algorithm such a construction can be found, how exactly it looks like and what type of redundancy it can produce.

The basic translation algorithm creates a *choice* construction from two constructs of the XSEM-H model: Content choices and node choices. The main difference between translations of these types of constructs is that a *choice* XML element translated from a node choice can have its *minOccurs* and *maxOccurs* attributes set.

The XML schema xs_{OUT} allows a *choice* construct to occur either under a *sequence* contruct or under another *choice* contruct. That means, in an arbitrary output of the basic translation algorithm, *choice* constructions can be mutually nested indefinitely.

The only limitation in such nestings is that if *choice* contruct is created from an XSEM-H node choice, it is not allowed to have another *choice* construct as its child.

Definition 12 *The graph of choices is a directed graph GC = (V,E), where V is a set of all* choice *constructions with* minOccurs *and* maxOccurs *attributes set to their default values from an output of the basic translation algorithm and* $E = \{(v1,v2) \mid v1, v2 \in V \wedge v1 \text{ is parent of } v2\}$. *A nesting of choices, denoted NC, is a connected component of GC.*

Obviously, each NC is a tree. The fact comes from an XML structure of a given output of the basic translation algorithm.

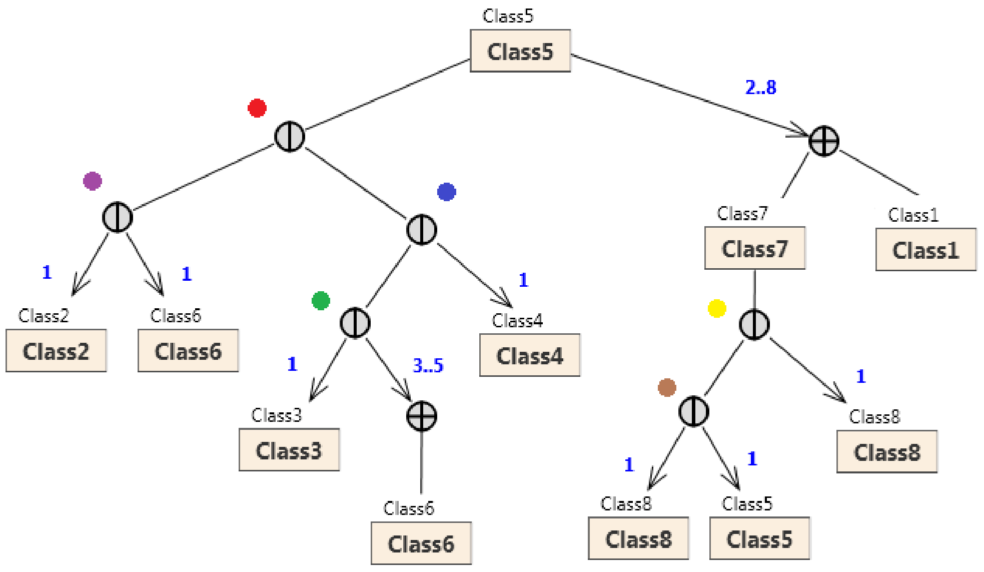


Figure 5.16: XSEM-H View, Redundancy in Nestings of Choices

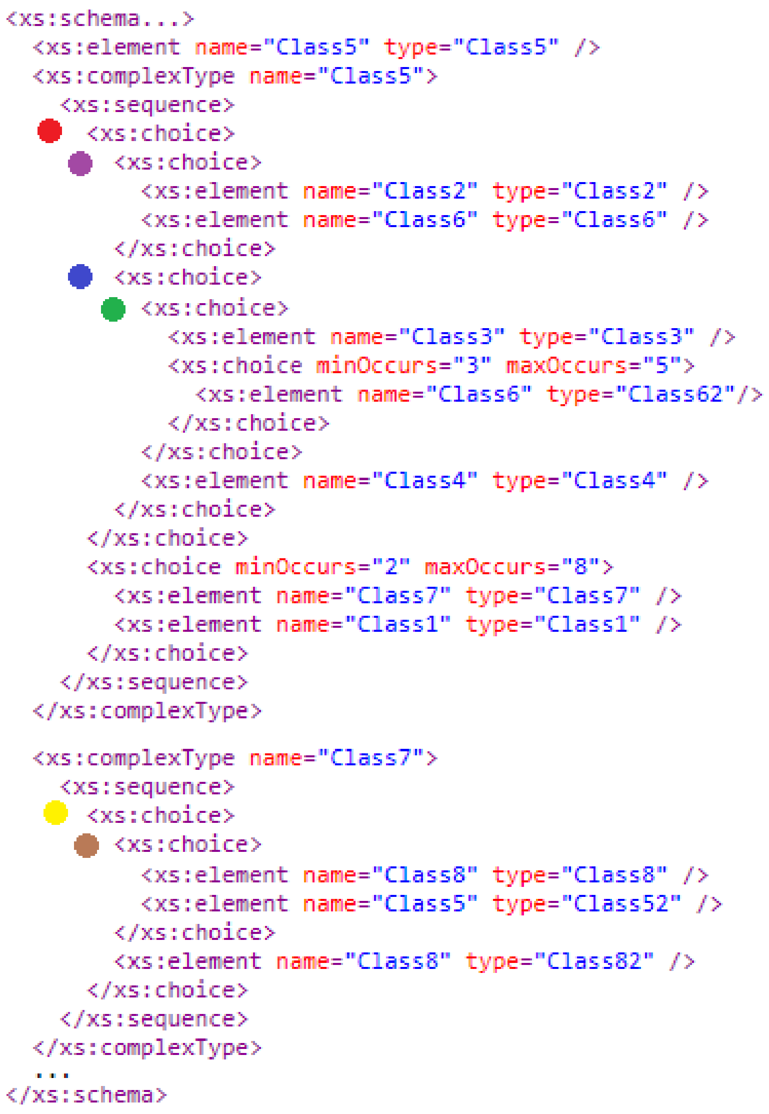
Let us illustrate Definition 12 in Figure 5.16. There is a sample XSEM-H view. A result of a translation of the XSEM-H view can be found in Figure 5.17. For clarity, empty complex types and attributes of *xs:schema* element are not included.

In the XSEM-H view, each content choice is displayed as a circle with one vertical line in it. Node choices are represented by circles with crosses.

A GC graph of the presented XML schema consists of six vertices, in the figures they are marked with a red, blue, violet, yellow, brown and green dot. The graph has two connected components, i.e. two NCs.

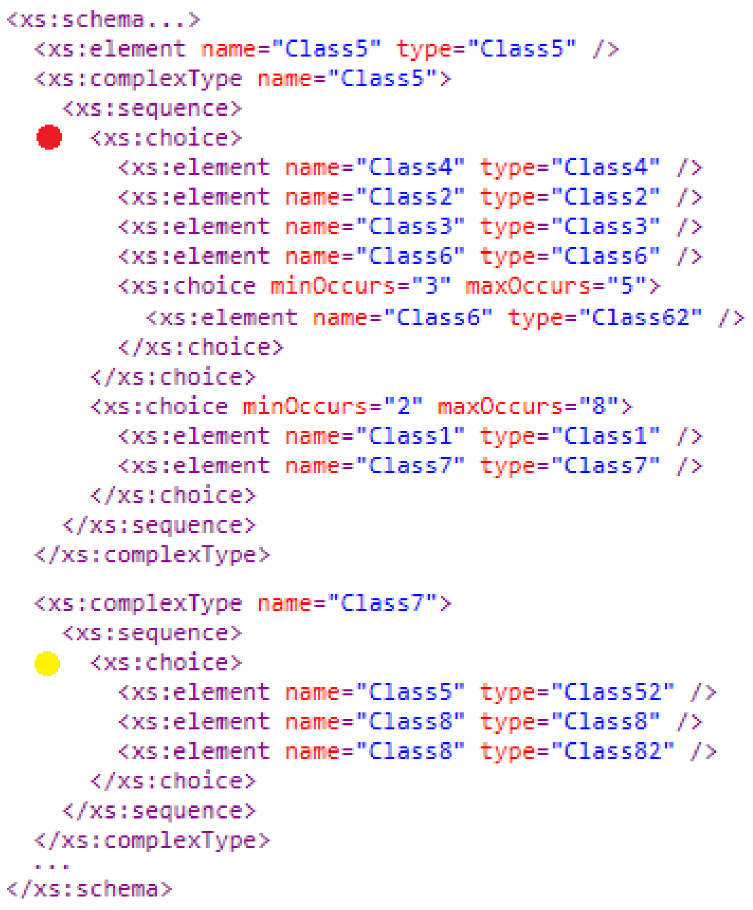
$$V_{GC} = \{\ v_{red},\ v_{blue},\ v_{green},\ v_{violet},\ v_{brown},\ v_{yellow}\ \}$$
$$E_{GC} = \{\ (v_{red}, v_{violet}),\ (v_{red}, v_{blue}),\ (v_{blue}, v_{green}),\ (v_{yellow}, v_{brown})\ \}$$
$$NC_1 = \{\ v_{red},\ v_{blue},\ v_{green},\ v_{violet}\ \}$$
$$NC_2 = \{\ v_{yellow},\ v_{brown}\ \}$$

Obviously, both the NCs from Figure 5.17 can be written in easier ways,



```
<xs:schema...>
  <xs:element name="Class5" type="Class5" />
  <xs:complexType name="Class5">
    <xs:sequence>
  ●   <xs:choice>
   ●    <xs:choice>
          <xs:element name="Class2" type="Class2" />
          <xs:element name="Class6" type="Class6" />
        </xs:choice>
   ●    <xs:choice>
     ●    <xs:choice>
            <xs:element name="Class3" type="Class3" />
            <xs:choice minOccurs="3" maxOccurs="5">
              <xs:element name="Class6" type="Class62"/>
            </xs:choice>
          </xs:choice>
          <xs:element name="Class4" type="Class4" />
        </xs:choice>
      </xs:choice>
      <xs:choice minOccurs="2" maxOccurs="8">
        <xs:element name="Class7" type="Class7" />
        <xs:element name="Class1" type="Class1" />
      </xs:choice>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Class7">
    <xs:sequence>
  ●   <xs:choice>
   ●    <xs:choice>
          <xs:element name="Class8" type="Class8" />
          <xs:element name="Class5" type="Class52" />
        </xs:choice>
        <xs:element name="Class8" type="Class82" />
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
  ...
</xs:schema>
```

Figure 5.17: XML Schema, Redundancy in Nestings of Choices



```
<xs:schema...>
  <xs:element name="Class5" type="Class5" />
  <xs:complexType name="Class5">
    <xs:sequence>
      <xs:choice>
        <xs:element name="Class4" type="Class4" />
        <xs:element name="Class2" type="Class2" />
        <xs:element name="Class3" type="Class3" />
        <xs:element name="Class6" type="Class6" />
        <xs:choice minOccurs="3" maxOccurs="5">
          <xs:element name="Class6" type="Class62" />
        </xs:choice>
      </xs:choice>
      <xs:choice minOccurs="2" maxOccurs="8">
        <xs:element name="Class1" type="Class1" />
        <xs:element name="Class7" type="Class7" />
      </xs:choice>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Class7">
    <xs:sequence>
      <xs:choice>
        <xs:element name="Class5" type="Class52" />
        <xs:element name="Class8" type="Class8" />
        <xs:element name="Class8" type="Class82" />
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
  ...
</xs:schema>
```

Figure 5.18: XML Schema, Redundancy in Nestings of Choices Output

so that the XML schema still describes the same set of XML documents. An example of an XML schema with optimalized *NC*s is illustrated in Figure 5.18.

Both the figures, illustrating the XSD documents, represent the same XML schema, althought in Figure 5.18 there are only two *choice* constructions without *minOccurs* and *maxOccurs* attributes - the red one and the yellow one.

The other *choice* constructs from Figure 5.17 don't add new information to the XML schema, because they are included in another *choice* constructs with the same meaning. Therefore, we consider the violet, blue, green and brown *choice* construction redundant.

We propose an algorithm for simplification of NCs, i.e. for elimination of redundant *choice* constructions from an output of the basic translation algorithm.

Let us suppose that we have a function, called *root()*, that gets any $C \in GC$ and gives $R \in GC$, which is the root of the nesting, which item C belongs to.

Then, an idea of the algorithm is very simple.

We get items of GC one by one and for each of them we get a collection of its children, which don't belong to GC (i.e. non-choice children). Then, we put the whole collection of nodes under the appropriate *root* element.

After processing of all the items of GC, we remove the non-root items from GC, as well as from the XML schema document.

Algorithm Analysis

The algorithm **terminates** for every input. This is obvious, because both finding roots and processing nodes of GC are very simple tasks.

A **correctness** comes from the fact, that for each item of an XML schema the algorithm either decreases a number of choice constructions, in which it is included, or it let the item's position unchanged.

Chapter 6

Conclusions and Future Work

In this thesis, we focused on a conceptual modeling for XML, on its purpose and specifics, and listed some of the most popular conceptual models. Then, in brief we described a conceptual model XSEM that was introduced in [4]. We said which items XSEM typically contains and introduced a conceptual modeling tool that implements it called XCase.

We described an algorithm for a translation of an XML schema from an XSEM model into the XML Schema language that was introduced in [4]. We called it the basic translation algorithm and discussed some of its limitations. Our observations led to the conclusion that the algorithm has a few significant and a number of less important limitations.

As an important limitation we introduced the fact that the basic translation algorithm didn't use well-known design patterns of the XML Schema language. Another significant limitation was ignoring grouping of entities that can possibly occur in the XSEM model. The last important limitation that was discussed refered to the fact that the basic translation algorithm produced output with lots of structural redundancies.

To be able to improve the algorithm, we defined formally how its output looks like. A set of all possible outputs has been precisely defined by an XML schema, that was presented usign platform-specific models of the XCase tool.

Motivated by the discovered limitations of the algorithm, we analyzed their possible solutions. Moreover, we proposed a concept of a translation profiling that helps users to influence a process of the algorithm through a special set of parameters.

An entire chapter of this thesis was devoted to a proposed solution of the problem with structural redundancies in an output of the basic translation algorithm. In the chapter we solved three most common types of redundancies.

We define a term *AP* and showed which *AP*s can be redundant and how to deal with them. Also, we proposed an algorithm for elimination of redundant attribute declaration and for simplification of nestings of choices and sequences.

Currently, we are working on an implementation of the proposed improvements. We are trying to include new functionalities, based on our ideas, into the XCase tool.

Bibliography

[1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C, September 2006. http:///www.w3.org/TR/REC-xml/.

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

[3] J. Clark and M. Makoto. RELAX NG Specification. Oasis, December 2001. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

[4] M. Necasky. Conceptual Modeling for XML. Ph.D. thesis, 2008. Department of Software Engineering, Charles University. http://www.necasky.net/thesis.pdf.

[5] B. Thalheim. Entity-Relationship Modeling: Foundations of Database Technology. Springer Verlag, 2000, Berlin, Germany. ISBN: 3-540-65470-4

[6] A. Badia. Conceptual Modeling for Semistructured Data. In Proceedings of the 3rd International Conference on Web Information Systems Engineering Workshops (WISE 2002 Workshops), p. 170-177. Singapore, December 2002.

[7] M. Mani. EReX: A Conceptual Model for XML. In Proceedings of the Second International XML Database Symposium (XSym 2004), p. 128-142. Toronto, Canada, August 2004.

[8] M. Mani, D. Lee, R. R. Muntz. Semantic Data Modeling Using XML Schemas. In Proceedings of the 20th International Conference on Conceptual Modeling (ER 2001), p. 149-163. Yokohama, Japan, November 2001.

[9] A. Sengupta, S. Mohan, R. Doshi. XER - Extensible Entity Relationship Modeling. In Proceedings of the XML 2003 Conference, p. 140-154. Philadelphia, USA, December 2003.

[10] J. Klímek, L. Kopenec, P. Loupal, J. Malý. XCase - A Tool for Conceptual XML Data Modeling. In Lecture Notes In Computer Science, Vol. 5968/2010, Advances in Databases and Information Systems, pp. 96-103, March 2010. ISBN 978-3-642-12081-7

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

[12] R. L. Costello. XML Schemas: Best Practices, 2003. http://www.xfront.com/GlobalVersusLocal.pdf

[13] A. Khan and M. Sum. Introducing Design Patterns in XML Schemas. Sun Developer Network Article, 2006.

[14] T. Bray, D. Hollander, A. Layman, R. Tobin, H. S. Thompson. Namespaces in XML 1.0 (Third Edition). W3C, December 2009. http://www.w3.org/TR/REC-xml-names/

[15] Universal Financial Industry Message Scheme (ISO 20022). http://www.iso20022.org/.

[16] C. K. Liu and D. Booth. Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. W3C, June 2007. http://www.w3.org/TR/wsdl20-primer/.

[17] R. Jelliffe. Schematron. May 2000. http://www.ascc.net/xml/resource/schematron/

[18] D. Lee and W. W. Chu. Comparative Analysis of Six XML Schema Languages. ACM SIGMOD Record, v.29 n.3, p.76-87, September 2000

[19] V. Apparao, S. Byrne, M. Champion, S. Isaacs. Document Object Model (DOM) Level 1 Specification. W3C, October 1998. http://www.w3.org/TR/REC-DOM-Level-1/

[20] B. Preneel. Analysis and Design of Cryptographic Hash Functions. February 2003.

[21] P. V. Biron, A. Malhotra. XML Schema Part 2: Datatypes (Second Edition). W3C, October 2004. http://www.w3.org/TR/xmlschema-2/.

Appendix A

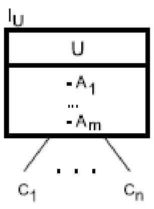
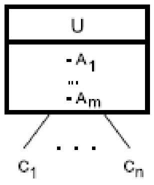
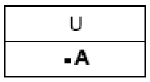
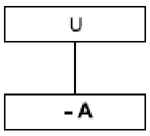
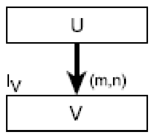
XML Schema Translations of XSEM-H Constructs

This appendix present a table of all XSEM-H model constructs proposed in Section 2.1.2. For each modeling construct there is its corresponding representation in the XML Schema language as described in Section 3.1.

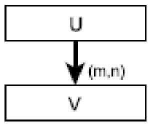
In the following table U denotes a node of an XSEM-H view, A is an attribute of U and C is a component from the content of U, i.e. C is an edge, attribute container, content container or content choice.

TN_U denotes a unique name assigned to U, called type name. XS_A is the translation of an attribute A and XS_C is the translation of C.

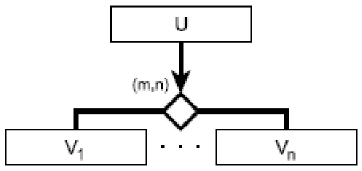
If C is an edge E from the content of U going to a node V without an element label, XS_C^{A-ref} is a reference to the attribute group translated from V. Otherwise, XS_C^{A-ref} is empty.

XSEM–H construct	XML Schema representation
(1) *Node with an element label*	
	`<xs:complexType name="`TN_U`">` `<xs:sequence>` $XS_{C_1} \ldots XS_{C_n}$ `</xs:sequence>` $XS_{A_1} \ldots XS_{A_m}$ $XS_{C_1}^{A-ref} \ldots XS_{C_n}^{A-ref}$ `</xs:complexType>` *if U is a root then* `<xs:element name="`l_U`" type="`TN_U`">`
(2) *Node without an element label*	
	`<xs:group id="`TN_U`-c">` `<xs:sequence>` $XS_{C_1} \ldots XS_{C_n}$ `</xs:sequence>` `</xs:group>` `<xs:attributeGroup id="`TN_U`-a">` $XS_{A_1} \ldots XS_{A_m}$ $XS_{C_1}^{A-ref} \ldots XS_{C_n}^{A-ref}$ `</xs:attributeGroup>`
(3) *Attribute* (N_A, dom_A) *of a node*	
	`<xs:attribute name="`N_A`" type="`dom_A`" />`
(4) *Attribute* (N_A, dom_A) *in an attribute container*	
	`<xs:element name="`N_A`" type="`dom_A`" />`
(5) *Edge going to a node with an element label*	
	`<xs:element name="`l_V`" type="`TN_V`" minOccurs="`m`" maxOccurs="`n`" />`

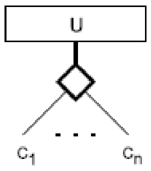
XSEM–H construct	XML Schema representation
(6) *Edge going to a node without an element label*	

	`<xs:group ref="`TN_V`-c"`
	`minOccurs="`m`" maxOccurs="`n`" />`

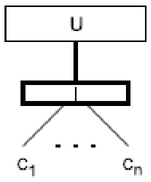
(7) *Edge going to node choice*	

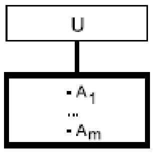
	`<xs:choice minOccurs="`m`"`
	`maxOccurs="`n`">`
	$XS_{E(V_1)} \ldots XS_{E(V_n)}$
	`</xs:choice>`
	if V_i *has an element label* l_{V_i}, $XS_{E(V_i)}$ *is* `<xs:element name="`l_{V_i}`" type="`TN_{V_i}`" />`
	if V_i *does not have an element label,* $XS_{E(V_i)}$ *is* `<xs:group ref="`TN_{V_i}`" />`

(8) *Content choice*	

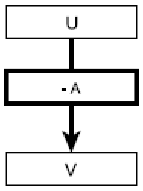
	`<xs:choice>`
	$XS_{C_1} \ldots XS_{C_n}$
	`</xs:choice>`

(9) *Content container*	

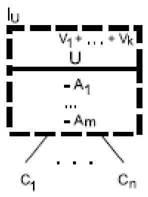
	`<xs:element name="`l`">`
	`<xs:complexType>`
	`<xs:sequence>`
	$XS_{C_1} \ldots XS_{C_n}$
	`</xs:sequence>`
	$XS_{C_1}^{A-ref} \ldots XS_{C_n}^{A-ref}$
	`</xs:complexType>`
	`</xs:element>`

(10) *Attribute container*	
	$XS_{A_1} \ldots XS_{A_m}$

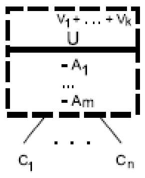
XSEM–H construct	XML Schema representation
(11) *Attribute container with an attribute* (N_A, dom_A) *and a connected edge* E	

	`<xs:element name="`N_A`">`
	  `<xs:complexType mixed="true">`
	    `<xs:sequence>`
	      XS_E
	    `</xs:sequence>`
	  `</xs:complexType>`
	`</xs:element>`
	where XS_E *is the translation of* E *according to (5-6).*

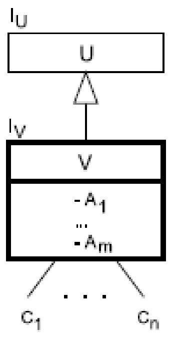
(12) *Structural representative with an element label*	

 l_U
$V_1 + \ldots + V_k$
U
- A_1
...
- Am
. . .
C_1
C_n
	`<xs:complexType name="`TN_U`">`
	  `<xs:sequence>`
	    `<xs:group ref="`TN_{V_1}`-c">`
	    `...`
	    `<xs:group ref="`TN_{V_k}`-c">`
	    $XS_{C_1} \ldots XS_{C_n}$
	  `</xs:sequence>`
	  `<xs:attributeGroup ref="`TN_{V_1}`-a">`
	  `...`
	  `<xs:attributeGroup ref="`TN_{V_k}`-a">`
	  $XS_{A_1} \ldots XS_{A_m}$
	  $XS_{C_1}^{A-ref} \ldots XS_{C_n}^{A-ref}$
	`</xs:complexType>`

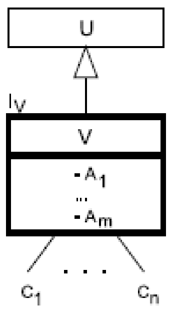
(13) *Structural representative without an element label*	

 $V_1 + \ldots + V_k$
U
- A_1
...
- Am
. . .
C_1
C_n
	`<xs:group id="`TN_U`-c">`
	  `<xs:sequence>`
	    `<xs:group ref="`TN_{V_1}`-c">`
	    `...`
	    `<xs:group ref="`TN_{V_k}`-c">`
	    $XS_{C_1} \ldots XS_{C_n}$
	  `</xs:sequence>`
	`</xs:group>`
	`<xs:attributeGroup id="`TN_U`-a">`
	  `<xs:attributeGroup ref="`TN_{V_1}`-a">`
	  `...`
	  `<xs:attributeGroup ref="`TN_{V_k}`-a">`
	  $XS_{A_1} \ldots XS_{A_m}$
	  $XS_{C_1}^{A-ref} \ldots XS_{C_n}^{A-ref}$
	`</xs:attributeGroup>`

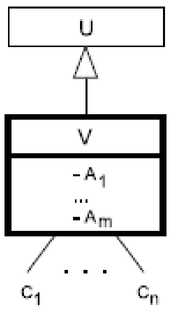
XSEM–H construct	XML Schema representation
(14) *Specialization V of a node U where U has an element label*	

 l_U
U
l_V
V
- A_1
...
- A_m
. . .
c_1
c_n
	`<xs:complexType name="`TN_V`">`
	  `<xs:complexContent>`
	    `<xs:extension base="`TN_U`">`
	      `<xs:sequence>`
	        $XS_{C_1} \dots XS_{C_n}$
	      `</xs:sequence>`
	      $XS_{A_1} \dots XS_{A_m}$
	      $XS_{C_1}^{A-ref} \dots XS_{C_n}^{A-ref}$
	    `</xs:extension>`
	  `</xs:complexContent>`
	`</xs:complexType>`
	
	if U is a root and V has a different element label from V then
	`<xs:element name="`l_V`" type="`TN_V`">`

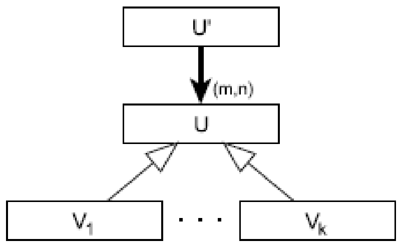
(15) *Specialization V of a node U where U does not have an element label and V does*	

 U
l_V
V
- A_1
...
- A_m
. . .
c_1
c_n
	`<xs:complexType name="`TN_V`">`
	  `<xs:sequence>`
	    `<xs:group id="`TN_U`-c" />`
	    $XS_{C_1} \dots XS_{C_n}$
	  `</xs:sequence>`
	  `<xs:attributeGroup id="`TN_U`-a" />`
	  $XS_{A_1} \dots XS_{A_m}$
	  $XS_{C_1}^{A-ref} \dots XS_{C_n}^{A-ref}$
	`</xs:complexType>`
	
	if U is a root and V has a different element label from V then
	`<xs:element name="`l_V`" type="`TN_V`">`

XSEM–H construct	XML Schema representation
(16) *Specialization V of a node U where U and V does not have an element label*	

	`<xs:group id="`TN_V`-c">`
	  `<xs:sequence>`
	    `<xs:group id="`TN_U`-c" />`
	    $XS_{C_1} \ldots XS_{C_n}$
	  `</xs:sequence>`
	`</xs:group>`
	
	`<xs:attributeGroup id="`TN_V`-a">`
	  `<xs:attributeGroup id="`TN_U`-a" />`
	  $XS_{A_1} \ldots XS_{A_m}$
	  $XS_{C_1}^{A-ref} \ldots XS_{C_n}^{A-ref}$
	`</xs:attributeGroup>`

(17) *An edge E going to a specialized node*	

	`<xs:choice minOccurs="`m`"`
	           `maxOccurs="`n`">`
	  XS_E $XS_{E(V_1)}$ $\ldots$ $XS_{E(V_k)}$
	`</xs:choice>`
	
	If U does not have an element label and is abstract, XS_E is empty. Otherwise, XS_E is the translation of E according to (5-6).
	
	If V_i has an element label, $XS_{E(V_i)}$ is `<xs:element name="`l_{V_i}`" type="`TN_{V_i}`" />`
	
	If V_i does not have an element label, $XS_{E(V_i)}$ is `<xs:group ref="`TN_{V_i}`-c" />`