A concept and implementation of higher-level XML transformation languages

Daniel Foetsch a, Elke Pulvermüller b, *  

 a Department of Computer Science, Christian-Albrechts-Universität zu Kiel, 24098 Kiel, Germany  
 b Department of Mathematics and Computer Science, University of Osnabrück, Albrechtstraße 28, 49069 Osnabrück, Germany

1. Introduction

System adaptation and integration (e.g. in a refactoring task) implies intensive transformation. In this paper, we concentrate on the system integration in the e-commerce domain. E-commerce is a field in which the integration and evolution requirements are central and particularly challenging. Therefore, they provide an ideal resource to derive requirements and solution concepts which may be transferred also to other domains.

E-commerce systems are constructed from a set of reusable services. In this, it has several similarities to the telecommunication domain. However, these services are frequently contributed and hosted from different vendors. The services often change due to technical progress and market competition. Examples for companies, which concentrate on the integration of e-commerce services to customer specific solutions, are Truition Inc. [1] and Intershop Communications AG. Truition Inc., for instance, offers on-demand solutions for specific e-commerce activities (out-sourcing of e-commerce services). Instead of building individual sales and marketing channels, Truition Inc. offers such services in different alternatives (e.g. product data transfer into customer specific online-shops or into market places like eBay). The customers connect their logistic systems, for instance, to the Commerce Management System of Truition and, thus, use the offered sale and marketing channels.

One of the most challenging drawbacks of such an approach is the fact that there is no one standard general for the integration of the different interfaces. Either no standard is available at all or there are multiple of them (e.g. BMEcat, EDIFACT or xCBL). Solutions which aim at connecting different e-commerce services have to provide a transformation service for ASCII text, Excel, XML and CSV (comma separated values), for instance. The latter of which, is the most frequently used format (53.7%) in the Truition Inc. business, for instance. Besides the different formats, it is a technical challenge to deal with usually frequently changing APIs (known as agile interfaces). This refers to both, the customers providing the (e.g. logistic) data as well as the sale and marketing platforms. The eBay interfaces, for instance, change approximately all 24 weeks. Examples of such changes are new attributes to classify the products or just new syntactical data structures.

Although the integration and adaptation problem is a key issue in the e-commerce market, the diversity of formats and frequency of individual changes have hindered an efficient automation. In our work, we aim at a contribution to closing this gap. In particular, we address the current problem of low-level transformations. Equal to high-level languages for defining and implementing the services or parts, respectively, we aim at suitable higher-level languages to define transformations which are performing the integration and adaptation tasks. As higher-level transformation languages depend on the domain, it is essential to support the definition of appropriate specific transformation languages.
For the realization of higher-level transformation languages we rely on XML. XML is often used as an intermediate language (e.g. also in the data transformation approach of Truition Inc.). XML (extensible markup language) [2] is a meta markup language recommended by W3C in order to create structured documents. Unlike plain text, it contains special tags which decompose the document into logical parts.

One of the most widely used XML transformation languages is XSLT recommended by W3C [3,4]. The origins of XSLT as a language, itself written in XML, lie in the functional paradigm, and in text-based pattern matching languages in the tradition of awk [5]. Accordingly, an XSLT program (called stylesheet) consists of a collection of template rules. Each template rule contains a match condition and a list of operators to construct parts of the target document. In short, if a node in the source document matches the template’s rule match condition, the template is instantiated and the operators specified within this template rule are executed. Additionally, a very simple function mechanism using named templates is offered.

Nevertheless, as a low-level transformation language, XSLT has drawbacks. XSLT does not offer operators to create constructs of a particular class of target documents. The base language supports only the creation of text literals, XML literals, and XML nodes like elements, attributes, comment etc. Even more significantly, XSLT completely ignores target language constraints. An XSLT processor can only ensure that the result XML document is well formed. It cannot guarantee the validity of a specific XML dialect with respect to the target schema. Other target formats are unsupported at all. This makes XSLT both a powerful and an error-prone low-level transformation language.

The remainder of the paper is organized as follows: The fundamental ideas of the generic XML operator hierarchy concept are presented in Section 2. It outlines the principles to the construction of higher-level transformation languages. Based on this introductory background, we propose a concrete operator hierarchy (i.e. a concrete higher-level transformation language) on top of the low-level transformation language XSLT in the following Section 3. After identifying several possible domain-independent higher-level operators for XSLT, we describe the definition, usage, and implementation of one example operator. This is followed by an example of a complete domain-specific transformation language. The concrete operators shall only demonstrate the realization of the operator hierarchy and prove the feasibility of the approach. The general concept is independent of the specific technology. Section 4 provides an overview about the developed XML Transformation Coordinator, XTC, which supports the processing of hierarchical transformation languages. Finally, we consider related work and conclude this paper.

### 2. Fundamentals

In this paper, we support higher-level transformation languages by building new higher-level operators on top of existing ones based on the generic XML operator hierarchy concept. The advantages of higher-level transformation languages are similar to those of general-purpose higher-level programming languages (e.g. Java). Modularizing the code reduces the redundancy and the risk of errors. Since the higher-level operators bear a close resemblance to everyday languages, the readability and maintenance of code is improved. The higher-level operators may increase the programmer productivity. Moreover, domain-specific operators, specified for a concrete XML dialect or another non-XML-based format, allow error messaging and, in general, validation at the domain level. Last but not least, domain-specific operators ensure that the result or at least parts of the result complies with the constraints of target languages.

The operator hierarchy concept picks up existing principles from software engineering. The basic idea is to construct a layered framework that enables the definition of higher-level operators on top of existing ones. This allows a bottom-up stepwise composition and adaptation of operators. A specific closed set of operators on one level forms a transformation language. The higher the level the more domain specific is the language or the operator, respectively. The operator hierarchy and the mapping between the levels bridges the gap between the low-level transformation language (e.g. XSLT) and high-level transformation languages (e.g. even more domain specific).

The starting point of the hierarchy is the lowest level of operators (cf. Fig. 1). This level comprises the so-called elementary operators offered by the underlying XML transformation technology. These could be the operators provided by the XSLT transformation processor, for instance. Elementary operators are used for fundamental operations on the document structure. On top of this elementary level, new levels are introduced in the operator hierarchy. These levels contain more complex operators. Although the figure ends on level 3 further levels may be added depending on the application domain.

We divide the set of new operators (on top of the elementary ones) in domain-independent and domain-specific operators. Domain-independent operators are predominantly defined to encapsulate more than one elementary operator for reuse. For instance, if an XML-based update language does not support the replacement of XML elements, a corresponding domain-unspecific replace operator could be composed from elementary insert and delete operators. Domain-specific operators are composed of elementary as well as domain-independent operators and can be particularly adapted to domain-specific languages and patterns. In contrast to domain-independent operators, these operators are basically intended for reuse in a certain application area. In principle, a more detailed differentiation as well as other kinds of classifications using other criteria is possible. The idea behind this principle to form higher-level transformation languages is similar to higher-level programming languages in general. Programs are compiled to lower-level statements (e.g. C++ is compiled to machine code). With the intermediate representations we have a similar hierarchy.

As shown in Fig. 1, the layered operator concept is implemented by applying a multi-level transformation process, which transforms operators from higher levels into operators from lower levels. If the transformation language uses itself XML to specify the transformation definitions, the language and the underlying transformation technologies can be reused for this level transformation. Some conceptual details of the operator hierarchy are elaborated in [6]. However, since more than one transformation must be performed in order to resolve higher-level operators, the hierarchical
approach may come with some additional costs regarding the performance of transformations. But this additional overhead has no impact on runtime performance. The transformation from higher to lower-level operators is performed only once at build-time.

3. XSLT operator hierarchy

As a basic principle, every language has its own operator hierarchy since higher-level operators completely depend on the expressive power of elementary languages or operators, respectively. For instance, an update language such as XUpdate [7] only supports inplace updates. As a consequence, it is impossible to build the same higher-level operators than using XSLT as underlying language. However, if two languages have the same expressive power, higher-level operators and, particularly, the newly built stand-alone domain-specific languages are portable.

In the following, we propose potential domain-independent and domain-specific operators for XSLT. In this case, XSLT is an example for a concrete elementary language providing concrete elementary transformation operators.

3.1. Domain-independent operators

The level of domain-independent operators offers a well-known set of enhanced functionality to perform common tasks. This large body of reusable code is provided to simplify the XSLT transformation programming. Its usage is not restricted to a specific domain but are, in contrast, cross-cutting to several domains. Various domain-independent operators can be built on top of elementary ones. In the following, we list some possible categories:

- **Math operators** provide facilities for mathematical computation, e.g. computation of the minimum, maximum, average, sum, or multiply of a node set; the highest or lowest node of a node set; absolute, ceiling, or floor value of a number node.
- **String operators** provide facilities for string manipulation, e.g. the replacement, splitting, concatenation, or alignment of one or more than one string.
- **Date and time operators** provide facilities for date and time formatting and conversion, e.g. returning a date, time, duration, year, month, name of month, days, name of days, minutes, seconds of one date/time string; adding or subtracting a date/time string with another one.
- **Set operators** provide facilities for nodes manipulation, e.g. the difference, distinction, intersection, union, equity, exceptions, or counting of two or more nodes.
- **Control operators** provide facilities for managing the control flow, e.g. functions, for-loop, for-each-group, while, or repeat operators.

To demonstrate the concept, we concentrate on two simple domain-independent operators. We start with describing the syntax and semantics of these operators followed by a discussion of their usage and implementation. This shows one complete realization and application step from low-level elementary operators to a higher-level domain-independent operator.

3.1.1. The function and the corresponding call-function operator

XSLT only offers a restricted function (call) mechanism. To realize a function in XSLT, a named template has to be declared which can be invoked by name. There are many restrictions to this concept. For instance, only one template with the same name is allowed. A parameter \( x \), passed to a named template that does not have a parameter for \( x \), is simply ignored without any warning. Particularly, the last restriction is error-prone. Applying the operator hierarchy concept, we can easily build a more flexible and safer function mechanism. By means of these new operators the developer can describe the transformation directly in terms of functions and function calls. The operators ease the transformation work for the developer by eliminating the burden to consider all these error-prone restrictions manually and, in consequence, support to reduce transformation errors. We call the corresponding operators function and call-function operator (the latter is used to call the defined function):

\[
\text{<ctrl:function name=qname>}
\text{<ctrl:param name=qname/>}
\text{<! - Content: template - -/>}
\text{</ctrl:function>}
\text{<ctrl:call-function name=qname>}\text{<varref>}
\text{<ctrl:with-param name=qname select=expression/>}
\text{<! - Content: template - -/>}
\text{</ctrl:with-param>}
\text{</ctrl:call-function>}
\]

The new function and call-function operator has each a required name attribute. The value of this name attribute is a qualified name.\(^\text{2}\) The parameters are defined in the same manner as in XSLT 1.0. The with-param is used to specify a parameter passing to a function. The required name attribute sets the respective name. Additionally, either a select attribute, containing an XPath expression [8], or a content can be applied to bind a value to this parameter. Similarly, a parameter can be defined in the function operator. However, in contrast to the named templates of XSLT, a corresponding function operator is identified by the required name attribute and the names of the parameters. Functions may be defined with the same name and different parameters and therefore supports function overloading.

The following simple example shows how to apply the new operators. Assuming, we have a list of orders as source, and we want to convert all time types in an application-specific format. This might be needed to connect two e-commerce services (e.g. the logistic of customer A with the eBay marketplace). The integration quality depends on a correct data exchange. As before, the stylesheet contains ordinary elementary XSLT operators and the new function operator and the respective call-function operator (cf. Listing 1). Their usage eases the definition of the transformation task and results in a more readable and a more concise representation of the transformation. The operators may be considered as a kind of small reusable transformation patterns or frameworks.

To generate the desired result, we need a level transformation which translates the domain-independent function operators into elementary XSLT operators (cf. Fig. 2). This is due to the fact that the underlying processors only understand the elementary operators; similar to a hardware processor which only understands machine code but no higher-level programming language. It is important that the function operators belong to a separate namespace because the corresponding level transformation file is identified by this namespace.

As shown in Fig. 2, the level transformation file is itself written in XSLT to reuse the XSLT processor. But, if more efficient, other specific XML transformation languages and respective processors may be applied, too. However, the ctrl.xslt level transformation

\(^\text{1}\) XSLT 2.0 uses sequences of nodes instead of the node set of XSLT 1.0.

\(^\text{2}\) To natively support higher-order functions, the name of the call-function operator may also be a variable reference to a function (name) supplied by a parameter of the surrounded function operator.
file supports and, thus, cross-cuts all domain-independent operators by providing facilities for managing the control flow of the transformation definitions. Line 2 of Listing 2 includes the level transformation definition of the function operators.

A possible level transformation definition, implementing a concrete function operator, consists of two templates (cf. Listing 3; extract of function.xslt). The content of the first one (line 6–13) contains exemplary XSLT operators ensuring the correct syntax of the function operator. Further checks may be specified in the same manner. Examples for further checks might be:

- if a function definition with the same name and parameters exists,
- if the parameters have different names, and
- if parameters have either a select attribute or a content.

Listing 1. Usage of the function operators

```xml
<OrderArray>
  <Order>
    <OrderID>2476679</OrderID>
    <OrderStatus>Completed</OrderStatus>
    <AdjustmentAmount currencyID="USD">0.0</AdjustmentAmount>
    <AmountSaved currencyID="USD">0.99</AmountSaved>
    <CheckoutStatus>
      <eBayPaymentStatus>PayPalPaymentInProgress</eBayPaymentStatus>
      <LastModifiedTime>20070914T18:31:35.000Z</LastModifiedTime>
      <PaymentMethod>PayPal/PaymentMethod>
      <Status>Complete/Status>
    </CheckoutStatus>
  </Order>
  ...
</OrderArray>
```

Listing 2. Level transformation definition of the ctrl stylesheet: ctrl.xslt.

```
01 <xsl:stylesheet
02   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
03   xmlns:ctrl="http://www.informatik.uni-kiel.de/Control">
04   <xsl:template name="convertTime">
05     <xsl:param name="oldTimezone"/>
06     <xsl:value-of select="concat(substring(SoldTimezone,'1', '4'), ' ',
07                      substring(SoldTimezone, '5', '8'), ' ',
08                      substring(SoldTimezone, '9', '12'))"/>
09   </xsl:template>
10   ...
11   <xsl:template match="/Order/CheckoutStatus">
12     <lastChange>
13       <xsl:call-template name="convertTime">
14         <xsl:param name="oldTimezone" select="LastModifiedTime"/>
15       </xsl:call-template>
16     </lastChange>
17   </xsl:template>
18   </xsl:stylesheet>
```

Fig. 2. XSLT level transformation (application of Fig. 1).
The content of the second one contains a variable declaration to create an unique name (lines 16–23) as well as XML literals and XSLT creating operators to transform the function operator into corresponding elementary XSLT operators (lines 24–38). The XML literals use an alias for a namespace (axsl) in order to avoid name-space conflicts with the XSLT namespace.

```
01<transform version="1.0"
xmlns="http://www.w3.org/1999/XSL/Transform"
xmlls="http://www.w3.org/1999/SXML/Transform"
xmlls:axsl="http://www.w3.org/1999/SXML/Transform/Alias"
xmlls:ctrl="http://www.informatik.uni-kiel.de/Control"
exclude-result-prefixes="ctrl">
02<namesspace-alias stylesheet-prefix="axsl"
result-prefix="xsl"/>
03<variable name="filename"
select="function.xslt"/>
```

Though the new version of XSLT (XSLT 2.0 [4] in conjunction with XPath 2.0 [10]) also allows the declaration of user-defined functions that can be called from any XPath expression. Nevertheless, there are some differences. Our function operator is uniquely identified by the combination of its name attribute and its parameter names. Thus, it supports function overloading. A function may be defined, for instance, with the same name and number of parameters but different named parameters. This approach differs from the XPath function call in that the values are passed by associating each one with a parameter name, instead of providing an ordered list of values. Furthermore, the structure of our call-function operator is reflected in the nesting of XML elements and their attributes. Hence, it can be generated from more higher-level operators easier than the compact textual representation of XPath. Due to the level transformation process, last but not least, our function operator natively supports the definition of higher-order functions. It is possible to pass a function (name) as a parameter to another callable function and then invoke it dynamically (cf. in contrast [11]).

However, the function operator is only one example for a domain-independent operator. There are many other (more complex) control operators that can be built on top of elementary ones. For instance, the hierarchy concept may be used to create a traditional for-loop operator with start and end attributes as well as an optional counter attribute. There is no comparable direct support in XSLT 2.0.

### 3.2. Domain-specific operators

Similar to the construction of above we build domain-specific operators. These offer facilities to create parts of a particular class of target documents. While XSLT is universal and thus only supports the creation of text literals, XML literals, and XML nodes such domain-specific operators provide higher-level and more specific (user-friendly) language constructs.

As outlined for the e-commerce domain this is a severe restriction as a large number of target document categories may be identified. The main distinction is between XML documents and non-XML documents. For XML documents many XML-based vocabularies, languages, and applications exist. This includes, for instance, WSDL, SOAP, ebXML, xCBL, XSD, RDF, OWL, and several more. Also for non-XML documents several non-XML-based vocabularies, languages, and applications exist. This includes, for instance, EDIFACT, CSV, PDF, LaTeX, DTD, and many more.

Applying the operator hierarchy concept, both categories of documents may be supported. Corresponding domain-specific operators may be defined by extending XSLT. This may even result in a completely new transformation language which consists of a closed set of such domain-specific operators. Since these operators are defined for a restricted domain, it is possible to check data consistency and ensure it to a certain degree. For example, the level transformation can generate control blocks which checks the type of selected data of the source document (see [13]).

1. In point of fact, we apply a bootstrapping process (i.e. applying our own operator hierarchy concept) by using XSLT-specific operators to generate elementary XSLT operators. In contrast to XML literals, this ensures the generation of syntactically correct XSLT operators.

2. The for-each operator of XSLT is not really a loop construct. To solve iterative problems you have to transform them into recursive problems. The resulting code could become easily unreadable.
and newlines[14].

values) the data fields and records are delimited using commas to separate its data items. For example, in CSV files (comma separated values configured in lines 3–5 ofListing 4.

The optional separator description, specified with the separator element, allows to define the specific delimiters. For instance, in CSV files (comma separated values) the data fields and records are delimited using commas and newlines [14].

3.2.1. XML2DSV transformation language

An XML2DSV program is represented by a dsv element in an XML document. The dsv element may contain a separator description and must contain a description for the data fields and records.

The optional separator description, specified with the separator element, allows to define the specific delimiters. For instance, the field-separator operator states the delimiter, and if the last field may or may not have an ending delimiter. If the separator description is omitted, the default setting corresponds to the values configured in lines 3–5 of Listing 4.

The data element is used to define the delimited data. The corresponding data selection is specified by adding record operators as children of the data element. The record operator contains a description of data fields. These are instantiated for each node identified by the XPath expression specified by the match attribute (line 14). With respect to these context nodes, the concrete data items are selected using the select attribute in the field operator (lines 15–18). If all data fields bind literal data characters (lines 9–12) the match attribute in the record operator may be omitted. There may be an optional header record (header operator) appearing as the first record of the DSV with the same format as ordinary records. The data, header as well as record, and field operators support an escaped-field attribute. It is set if a character, specified in the field-escape operator, is used to enclose fields or operators containing references (cf. Listing 5). In the static mode, the specific delimiters are represented by text literals instead of variable declarations and variable references.

In both modes the selected data of the source document are checked. The test in the last template in Listing 5, for instance, ensures that the selected data do not contain escape characters. It avoids the common problem of delimiter collision. Further checks may be specified in the same manner. This named template and the respective calls are generated from the presented higher-level function operators.

Due to space limitations, we omit the implementation of the level transformation. Its principle is the same as for level transformations of domain-independent operators.

4. XML Transformation Coordinator (XTC)

To automate the multi-level transformation of the operator hierarchy concept, we developed the XML Transformation Coordinator (XTC) [12]. This provides a supporting environment for applying hierarchical transformation operators.

XTC consists of (cf. Fig. 3):

- a core system managing the transformation process,
- an operator library storing APIs for transformation operators,
- a level library as storage for the level transformation definitions implementing the operator APIs, and
- the adapters integrating transformation engines into the core system.

XTC may be invoked from the command line or from a Java application. The core system controls and manages the input, the output and the overall multi-level transformation process. Fig. 4 schematically illustrates this transformation process. In the first step the core system analysis the input: the source to be transformed and the transformation definitions. The transformation definitions are built by using either elementary operators of a basic transformation language or higher-level transformation operators specified in the operator library of XTC. In the preprocessing the core system checks if the transformation definition contains only elementary operators. In this case the appropriate transformation engines or their adapters, respectively, are invoked controlled by the namespace declaration. In the other case, i.e. higher-level operators are used in the transformation definition, the referenced level transformation definitions in the level library are applied to transform the higher-level operators into lower-level ones. The core system identifies the necessary transformation engine for this level transformation. This procedure is repeated as long as the result still contains higher-level operators. The recursion ends if the transformation definition itself may be processed by an available transformation engine. This hierarchical transformation process reflects the operator hierarchy.

Several transformation engines might be invoked by the core system due to two reasons: (1) the input is heterogeneous and may be only performed by different transformation engines and (2) level transformation definitions may be specified with different basic transformation languages. Furthermore, the operator hierarchy concept can be built on different basic transformation languages using one coordination environment.

For some transformation engines adapters are necessary for the communication between core system and engine. Moreover, these
Adapters may provide additional functionality. For instance, they may consist of validation engines to ensure that the transformation conforms to given grammar or semantic constraints.

4.1. XTC and the application example

Referring to the examples in Section 3 we may assign the code excerpts to the XTC architecture. However, the examples merely show the internals of XTC but demonstrate instead the application viewpoint of a XTC user. Section 3.1 lists a set of domain-independent operators. Some of which are detailed (e.g. the function operator given with its usage syntax). These operators may be stored in the operator library as operator pattern for future reuse.

Listing 1 represents the source and the extended XSLT transformation definition which form the input to XTC. The example does
not detail whether the input is given by command line or by the invocation of a Java application. The section called “Stylesheet” within Listing 1 is the concrete transformation definition using a higher-level operator. This stylesheet could also be used as a new higher-level transformation operator if similar transformations are expected to occur frequently. In this case a new operator specification must be registered in the operator library and a corresponding level transformation definition is transferred to the level library.

Listing 3 depicts a level transformation example located in the level library of XTC. Written once it may be reused for further level transformation tasks. All transformation definitions using the new domain-independent function operators can be processed with the help of this new level transformation definition resulting in a format which can be processed by the corresponding transformation engine. XTC automatically detects new transformation operators in the input and checks for applicable level transformations (stored in the level library). This is a recursive process.

Listing 2 gives a glance at some internal XTC components. The XSLT excerpt controls the application and execution of different level transformations for control operators provide facilities for managing the control flow, i.e. those stored in the level library. Processing the input (the transformation definition and the source), the level transformations stored in the level library XTC forwards its result to the transformation engine. In the example, the engine is an XSLT processor.

5. Related work

The paper presents a concrete realization and application of the operator hierarchy concept to define higher-level transformation languages. The usage of higher-level languages for implementing the system parts (e.g. components, objects) is not sufficient. Facing an increasing integration and evolution pressure, transformations become a major element in the system development. The following outlines related work which is also aiming at providing more convenient (higher-level) languages and tools to express the transformations.

While there exist numerous approaches to modularize low-level transformation operators for the purpose of clarity and reuse (e.g. the \texttt{include} and \texttt{import} operator in XSLT) extended and XSLT specific reuse mechanisms may be found in the EXSLT community initiative [15]. It proposes a number of modules including named templates to provide extensions to XSLT. These named templates are comparable to our domain-independent operators. The main difference between both concepts is that higher-level operators are like pre-defined language instructions. Therefore, the operators offer an appropriate XML-based notation. Furthermore, the syntax of operators can be validated and corresponding error messages may be thrown. In XSLT, for instance, it is not an error to pass a parameter to a template that does not have a respective parameter definition; the passed parameter is ignored without any warning.

Related work may also be found in further specific extensions and elementary operators (even those already available in XSLT). Closely connected to XSLT is XPath. XPath is a language for selecting specific nodes or node set from XML Documents. XPath offers a pre-defined set of basic operators. Indeed this pre-defined set does not cover all requirements. Thus, in the new version of XPath (in combination with XSLT) it is possible to declare user-defined functions. However, as mentioned above, a flexible and extendable hierarchy of highly reusable operators is not considered. Some XSLT processors such as Saxon [16] offer extension functions and extension operators for XSLT. For instance, the \texttt{saxon:while} operator performs a traditional while loop. The content of the \texttt{saxon:-while} operator is carried out as long as some conditional expression evaluates to true. But these operators are proprietary extensions of the standard. The respective XSLT programs are not portable. They cannot be processed by other XSLT processors.

Currently, higher-level transformation languages are directly implemented using general-purpose programming languages with APIs such as DOM or its derivates (JDOM, dom4j, etc.). The XOP-T composition concept [17], for instance, follows this approach. There, domain-specific operators are designed to process Java code after a transformation to JavaML [18], an XML representation of Java. Other top-down approaches (e.g. [19–21]) develop domain-specific transformation languages and map them on XSLT also using general-purpose programming languages. All these approaches consider only a single level of transformation operators in contrast to the layered hierarchy concept we propose.

The operator hierarchy, however, is not tied to a specific technical implementation. Though presented for the XML-specific transformation language XSLT in this paper, the concept is generic and, therefore, may be also applied to other languages. For instance, program transformation languages (such as ASF+SDF, DMS, Kids, Stratego, Tom, or TXL) may be used as underlying transformation technology, too.
OMG’s QVT (query/view/transformation) standard [22] and other model transformation languages are on another technological space than our transformation concepts. These existing transformation languages are, therefore, not directly comparable. However, the usage of the operator hierarchy concept may support the realization of higher-level model transformations. They may be realized by means of model-specific transformation operators.

### 6. Summary and outlook

In this paper, we propose the development of higher-level transformation languages. To demonstrate the realization of such a higher-level transformation language construction, we apply the XML operator hierarchy concept to the transformation language XSLT. XSLT is only one example which may be employed to provide elemental transformation operators. A specific set of higher-level operators forms the higher-level language. They are defined on top of the elementary operators. The layered concept allows the definition of new higher-level operators without changing the underlying XSLT language and respective processors.

We identified some examples of potential domain-independent and domain-specific operators. Domain-independent operators, composed of elementary and domain-independent operators, are specialized for a certain target dialect or format. The layer concept may be extended with additional layers depending on the application domain. We demonstrate the definition, usage, and implementation of higher-level operators using corresponding examples. XTC (XML Transformation Coordinator) provides support in developing and processing the hierarchical operators.

The experience with an increasing integration and evolution challenge in the e-commerce domain reveals that transformation has been neglected in the past with respect to the language level. Transformation should become a core element in the system development. It should be supported by powerful and suitable language constructs. At the same time, flexibility is required to react on changing or new application domains (and their specific requirements for suitable transformation languages). Modern paradigms and concepts like object-orientation, components, aspect-orientation or patterns for general-purpose programming languages are still missing for transformation languages. The experience in our work has shown that the mapping of such higher-level concepts to transformation languages is not straightforward. Even more, a pure mapping and knowledge transfer from the general-purpose programming languages may be inappropriate for transformation languages although the investigation is worthwhile.

In general, the proposed composition concept, relying on transformation itself, can be applied to several specific XML-based languages both transformation languages (STX [23], fxt [24], XML Script [25], etc.) and update languages (XUL [26], XUpdate [7], etc.). However, future research has to elaborate the expressive power of the specific XML-based languages and the effects on the expressive power of higher-level transformation definitions.

Further extensions to the work presented may be in the field of design guidelines to support the decision which elementary operators may be encapsulated in domain-independent operators and what measures may be employed to find a well-balanced size of the operators. Although we distinguish between domain-independent and domain-specific operators the area of domain-specific languages is not yet fully elaborated in this context. The presented approach provides the technological concept and is also able to deal with non-XML domain-specific languages already. However, the development and selection of a suitable domain-specific language is beyond this work. The integration of ontologies might be one way to extend our work in this direction.

Measurements may further improve the work. Subsequent work can measure the amount of reuse and development time reduction, the transformation result quality (e.g. in terms of readability or result performance) or the effort on the transformation speed, for instance. However, such measurements are not straightforward but will require a careful design of the measurement settings instead as the comparison results depend on the varying implementation style.

### References