A Domain Specific Language for UN/CEFACT's Core Components

Philipp Liegl, Dieter Mayrhofer
Vienna University of Technology
Favoritenstrasse 9-11/188
A-1040 Vienna, Austria
{liegl, mayrhofer}@big.tuwien.ac.at

Abstract

In order to overcome the heterogeneities of different business document standards the United Nations Center for Trade Facilitation and Electronic Business (UN/CEFACT) has released the Core Components Technical Specification (CCTS). Core components are reusable building blocks for assembling business documents in an implementation neutral manner. However, core components are standardized without considering a specific implementation format and thus no tool integration is possible. Currently a syntax specific solution for core components, based on the Unified Modeling Language (UML), is provided with the UML Profile for Core Components (UPCC). In this paper we circumvent the complex UML meta model and provide a dedicated core component modeling environment based on a Domain Specific Language (DSL). Thereby, core component models are assembled on a conceptual level. In a next step the conceptual document model is used for the generation of domain specific artifacts. Our DSL based solution provides in situ validation of conceptual core component models and the flexible generation of deployment artifacts such as XML Schema definitions, used for the definition of interfaces in a service oriented environment.

1. Introduction

In the past several business document standardization initiatives have been found, driven by different motivations and needs [5] [7]. Due to the abundance of standards, interoperability issues between the different standard definitions are rather the rule than the exception. A single solution, provided and maintained by a global initiative would help to overcome the limitations of industry specific and incompatible standard definitions. Thereby, a set of reusable components is defined by a global consortium, reflecting a superset of all different requirements. Industry partners may retrieve these components from a central library, and tailor them to their specific needs. The tailoring mechanism is strictly limited to a derivation by restriction and thus every specialized component may be traced back to the original generic component definition. In 1999 UN/CEFACT (United Nations Center for Trade Facilitation and Electronic Business) started their work on such a standardization approach which became known as the Core Components Technical Specification (CCTS) [14].

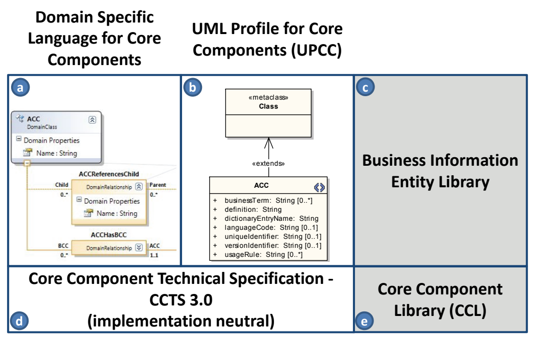
Domain Specific Language for Core Components
UML Profile for Core Components (UPCC)
a
b
c
ACC
«metaclass» Class
«extends»
Business Information Entity Library
Core Component Technical Specification - CCTS 3.0 (implementation neutral)
d
Core Component Library (CCL)
e

Figure 1: Overview of the core component initiative

The core component initiative aims at the definition of context independent building blocks, used for creating business documents. As shown in Figure 1d the Core Component Technical Specification (CCTS) is the foundation of the entire standardization initiative. Within the CCTS the basic concepts and rules for core components are defined in an implementation neutral manner. Thus, integration into modeling environments and other tools is difficult. In order to overcome this limitation UN/CEFACT started to work on a UML Profile for Core Components (UPCC) definition. As shown in Figure 1b the UPCC builds directly upon the CCTS by transforming the implementation neutral core component concepts into a UML representation. However, as we will pinpoint in this paper, the UML based approach has several shortcomings, since the generic modeling language UML has to be tailored to the specific needs of business document modeling. In our solution as shown in

Figure 1a we build on an emerging and promising technology called Domain Specific Language (DSL). Using a Domain Specific Language it is possible to build a dedicated and streamlined modeling language for any given application scenario and context. Thereby, no restrictions as for example with UML apply. Thus, with our DSL based approach the user is provided with an easy to use modeling language, which is fully core component compliant.

The remainder of this paper is structured as follows: In Section 2 the basic concepts of the Core Component Technical Specification (CCTS) are introduced, necessary for further conception of the Domain Specific Language. The basic concepts of Domain Specific Languages are introduced in Section 3. In Section 4 we elaborate on the concepts of our Domain Specific Language for Core Components together with an accompanying example. The major advantages of a DSL, compared to UML based approaches are outlined in Section 5. Section 6 gives an overview of related work in the area of domain specific languages and finally Section 7 concludes the paper.

2. Core Components at a glance

Figure 2 gives an overview of the basic concepts of the Core Components Technical Specification. We distinguish between two major paradigms: Core Components (CC) and Business Information Entities (BIE). Core components are context-independent building blocks for the creation of business documents. In order to contextualize a given core component to the specific needs of a given business domain, the concept of business information entities (BIE) is used. Business information entities are derived from core components by restriction and may be used in a certain business domain. Due to the derivation-by-restriction mechanism a business information entity must not contain any attributes, which have not been defined in the underlying core component. In a nutshell, business information entities are the same as core components except that they a) are based on core components and thus restrict an existing core component b) use a different naming convention in order to facilitate distinction from core components, and c) are used in a specific business context. Consequently core components represent the common semantic basis for all business information entities.

Having these concepts at hand it is possible to model business documents first on a conceptual, syntax neutral, and platform-independent level using core components. The created core component libraries as shown in Figure 1e may then be used to create different libraries of business information entities (cf. Figure 1c). Interoperability between different business information entities is guaranteed, since all business information entities are based on the same core components.

Core Components. The basic concept behind core components is the identification of objects and object properties. We distinguish between two different object properties: simple object properties (e.g. date, name, age) and complex object properties (references to other objects). An object is represented using an aggregate core component (ACC). Aggregate core components represent an embracing container for simple object properties. On the left hand side of Figure 2 the ACC `Person` includes two simple object properties - `date of birth` and `first name`. Simple object properties are referred to as basic core components (BCC). Each simple object property has a data type known as core data type (CDT). For example, the core data type of `date of birth` in the aggregate core component `Person` in Figure 2 is `date`. Core data types define the exact value domain of basic core components. In order to denote dependencies between different objects, the concept of association core components is used (ASCC). In Figure 2 the ACC `Person` is connected to the ACC `Address` using the association core component (ASCC) `Private`. Thus,

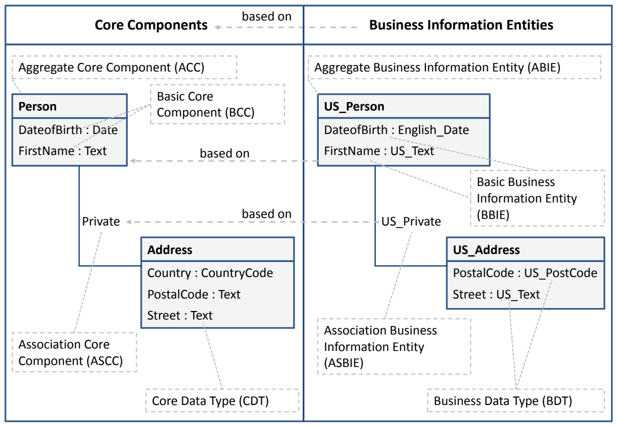

Figure 2: Overview of basic core component concepts

`Address` is a complex object property of the ACC `Person` and indicates the fact, that a person has a private address.

Business Information Entities. If core components are used in a certain business context, they become so called business information entities. Similar to the concept of core components, business information entities distinguish between objects and properties of objects. An aggregate business information entity (ABIE) is used to aggregate simple object properties. Simple object properties are denoted using the concept of basic business information entities (BBIE). On the right hand side of Figure 2 the ABIE `US_Address` aggregates the two BBIEs `postal code` and `street`. Please note, that the business information entity `US_Address` does not contain all attributes of the underlying core component `Address`. Thus, the ABIE restricts the underlying ACC. The value domain of basic business information entities is defined using the concept of business data types (BDT). The BBIE `postal code` contained in the ABIE `US_Address` is of type `US_PostCode`. Associations between different business information entities are denoted using the concept of association business information entities (ASBIE). Please note, that a business information entity must contain the name of the underlying core component. However, so called qualifiers may be used in order to facilitate the distinction between core components and business information entities. E.g the qualifiers `US_` and `English_` are used in Figure 2 for all business information entity definitions.

Data Types. As already previously outlined, the value domains of basic core components and basic business information entities are defined using the concept of data types. In general we distinguish between two different data type families - context independent core data types (CDT), used for core components and context specific business data types (BDT), used for business information entities. On the left hand side of Figure 3 the core data type `Identifier` is shown. Data types consist of two elementary concepts - exactly one content component (CON) and multiple supplementary components (SUP). Thereby, a content component holds the actual information such as '12' and supplementary components provide additional meta information about the content component (e.g. type of measure = temperature, measuring unit = Fahrenheit). Similar to core components, core data types are defined on a context neutral level. Core data types are exclusively used in order to set the value domain of basic core components.

We already mentioned, that a core component becomes a business information entity, if it is used in a certain business context. During the transformation of a core component to a business information entity, the core data types of the core component become business data types. The business data type `Language_Identifier` on the right hand side of Figure 3 does not contain any supplementary components, which have not been defined in the underlying core data type `Identifier` on the left hand side. Thus, the same derivation-by-restriction mechanism as used with core components and business information entities, is also used for core data types and business data types. Similar to business information entities the concept of qualifiers is used in order to distinguish business data types from core data types. The business data type on the right hand side of Figure 3 uses the qualifier `Language_`.

In order to specify the value domain of a content component or a supplementary component the concept of primitive types (PRIM) and enumerations (ENUM) is used. Primitive types are shown at the bottom of Figure 3, representing the most common data types. The concept of enumerations is used in order to restrict a certain primitive type to an allowed set of values.

Core Component Library. Since it is imperative that

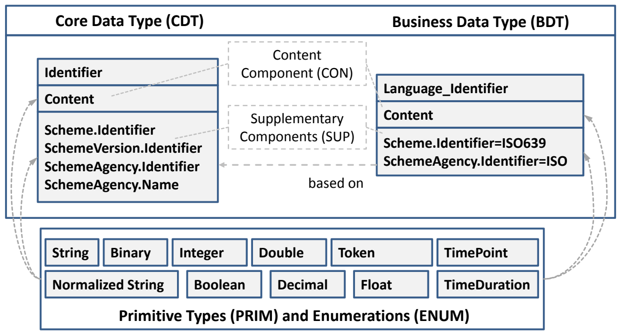

Figure 3: Core Data Type Overview

all business information entities are based on a core component, a consolidated library of predefined core components must be provided. A global library of core components is currently developed and maintained by UN/CEFACT and has become known as Core Component Library (UN/CCL) [15]. The core component library is shown in Figure 1e. Business document modelers may search and retrieve core components from the core component library and tailor them to the specific needs of a certain application domain. These newly created business information entities are stored in business information entity libraries for further use (shown in Figure 1c). Furthermore, anybody interested in creating a new core component may submit a new core component definition to UN/CEFACT, where the harmonization and standardization of core components is organized. The core component definitions in the Core Component Library represent the common semantic basis of all business information entities exchanged among business partners.

A UML Profile for Core Components. Core components are currently defined in an implementation neutral manner, and thus integration into modeling tools is difficult. In order to apply the core component concepts as a means of modeling business documents, UN/CEFACT has released the UML Profile for Core Components (UPCC), that allows using core components with standard UML modeling tools. For a detailed discussion of the UML Profile for Core Components see [6]. Using the UPCC a business document modeler first assembles core component models on a conceptual UML based level. Using a pertinent generator, the conceptual model is transformed into an XML Schema representation for deployment to an IT system. However, the UML Profile for Core Components has a set of shortcomings which originate from the fact, that the Unified Modeling Language is a general purpose language. In order to tailor the generic UML to the specific needs of the Core Component Technical Specification, certain workarounds are necessary. Furthermore, the implementation of XML generators on top of existing UML tools is a difficult and error prone task. In the following Section we introduce our proposed solution based on a Domain Specific Language (DSL) and specifically elaborate on the advantages of a DSL based core component implementation compared to a UML based solution.

3. Domain Specific Languages

A domain specific language (DSL) is a language tailored to a specific domain for solving a wide range of different problems. In general we distinguish between textual and graphical DSLs. A textual DSL represents domain specific characteristics and relationships between the different characteristics in a textual manner. Compared to a general purpose language such as C or Java, which address problems of a wide area, a textual DSL focuses on a well defined problem area. Since textual DSLs are less powerful than graphical DSLs, we do not consider them in our approach. Compared to a textual DSL, a graphical DSL provides an intuitive and error insusceptible approach to create valid concepts of the domain. DSL models may be used to derive machine processable artifacts. These generated artifacts include program code or descriptive definitions for service oriented architectures such as Web Service Definition Language (WSDL) or XML Schema files.

The graphical DSL used in our solution is based on Microsoft Visual Studio DSL Tools, which was first introduced in version 2005 of Microsoft Visual Studio. Visual Studio DSL Tools enable the graphical definition of a DSL directly in Microsoft Visual Studio. Figure 4 gives an overview of our approach. First, a DSL modeler creates the DSL definition together with custom code on the meta level. In case of our core component approach the custom code includes definitions, necessary for the validation of the core component model. In a second step the DSL definition from the meta-level is used to generate a core component designer. Since the core component designer has been generated out of the domain specific language, it contains only those modeling elements, necessary for the assembly of a business document model. A business document modeler uses the core component designer to model core component compliant business document models as shown in the center of Figure 4. Using the built-in template mechanism of

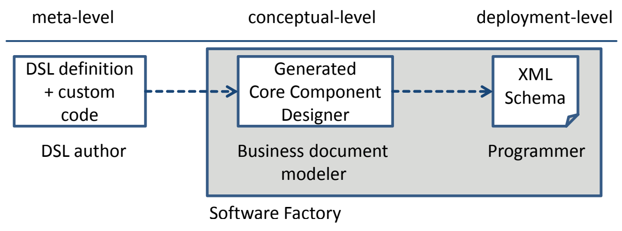

Figure 4: DSL approach

the Microsoft DSL, the DSL based core component model may be transformed to XML schema artifacts for the deployment level. Thus, the DSL approach perfectly fits our requirements to first define a business document model on a conceptual level and secondly use the conceptual model to derive XML Schema artifacts for a service oriented environment. Furthermore, Software Factories may leverage the use of DSLs. A Software Factory provides a guidance for the modeler throughout the whole modeling process from creating the model to generating the artifacts. Software Factories may also include smart wizards, which assist the modeler and help to automate repetitive tasks. Wizards may for example be used to derive a business information entity from a core component definition. Additionally, the validation component allows to check the correctness of a model, making the approach suitable for inexperienced business document modelers as well.

In the following we briefly introduce the meta-modeling environment for creating a Domain Specific Language in Visual Studio (meta-level in Figure 4). The conceptual level, where the actual core component models are assembled using the DSL generated core component designer, is introduced in Section 4. As seen in Figure 5, the DSL Tool's meta modeling canvas consists of two swimlanes: *classes and relationships*, and *diagram elements*. The entire model shown in Figure 5 is referred to as *domain model* and defines the logical structure of the DSL. A domain model, among other things, consists of *domain classes* (round-corned squares) and *domain relationships* (squares).

In a nutshell, domain classes define the classes of elements which may occur in an instance of the DSL. Domain classes may have attributes, which are called *domain properties*. Domain properties consist of a name and a data type. Relationships among domain classes are defined using the concept of domain relationships. A domain relationships may either be an *embedding relationship* (shown as a solid line) or a *reference relationship* (shown as a dashed line). Embedding relationships embed the element of the target domain class into the elements of the source domain class. Thus, if the source element is deleted, the embedded target elements are deleted as well. In contrary, reference relationships do not delete referenced elements if the source element is deleted. This will prove to be of particular interest when modeling core components with the DSL. The domain class `CCModel` on the upper left hand side of Figure 5 is the root class of the DSL. There may be at most one root class in a given DSL. In the instance model the root class will represent the parent element for all other classes and their instances.

On the right hand side of Figure 5 different *diagram elements* of the DSL are shown. Diagram elements are used to define the shape of domain classes and domain relationships in the model instance, which will be generated out

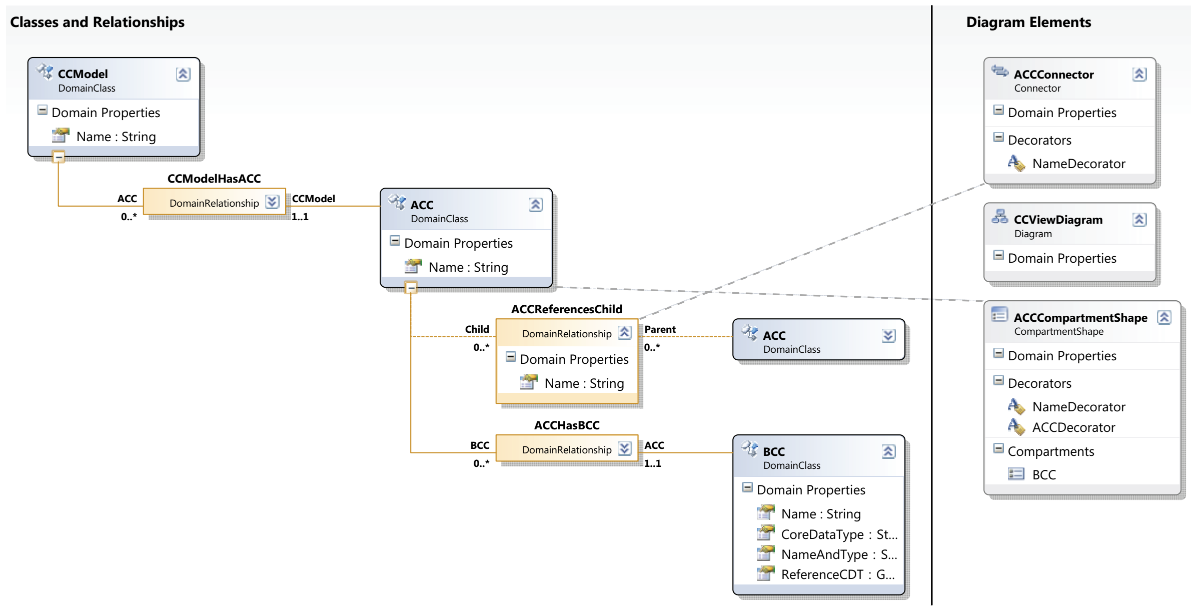

Figure 5: Domain model for Core Components

of the DSL definition. Thus, any shape of choice may be assigned to an arbitrary domain class or domain relationship. Thereby, a DSL based approach goes beyond classical modeling approaches such as Event Driven Process Chains (EPC) or Unified Modeling Language (UML), where the modeler is bound to a confined set of elements and their assigned shapes. E.g. in most of the UML modeling environments a UML class will always be represented as a rectangle, although the meta model of UML would in principle allow any pertinent shape.

4 The Core Component DSL

For our solution we created six DSLs namely for core components (CC), business information entities (BIE), core data types (CDT), business data types (BDT), primitive types (PRIM), and enumerations (ENUM). Due to space limitations we will only dwell on the concepts of the core component DSL. After we examine the meta model of the DSL we provide an example DSL instance, representing a purchase order document. The required steps for creating a core component model and deriving a business information entity model from it will be explained. Furthermore, we elaborate on the technical realization of important features such as model transformation, artifact generation, validation, and serialization.

The Visual Studio DSL Tools provide a graphical interface to conveniently assemble a specific DSL. Figure 5 shows the DSL meta model for core components. The element `CCModel` represents the root of the model, including a name property. `CCModel` may have any number of aggregate core component (ACC) elements, defined through the `CCModelHasACC` domain relationship. Any ACC may reference other ACCs defined through the `ACCReferencesChild` domain relationship, which also contains a name property. This domain relationship represents an association core component (ASCC). An ACC element may contain multiple basic core component (BCC) elements, defined through the `ACCHasBCC` domain relationship. A BCC element has four distinctive properties: *Name, CoreDataType, NameAndType*, and *ReferenceCDT*. *Name* defines the name of the BCC and *CoreDataType* is the name of the associated core data type. *ReferenceCDT* contains the Globally Unique Identifier (GUID) of the referenced core data type. Since the DSL Tools in the current version do not support cross-model references natively, we decided to use GUIDs in order to make cross-model references as described in [4]. *NameAndType* is for presentation purposes only and contains the name of the basic core component together with the name of its associated core data type.

As shown on the right hand side of Figure 5 there are three diagram elements in the core component DSL: `ACCConnector`, `CCViewDiagram`, and `ACCCompartmentShape`. An `ACCConnector` is mapped to the `ACCReferencesChild` domain relationship. Thereby, the connector shape associates two `ACCCompartmentShapes`. The shape prescribes, that on the source side the tip of the connector represents a diamond and on the target side the tip is simply a solid line. Additionally, the target side is annotated with the name of the assocation core component (ASCC), represented by the `NameDecorator` property in `ACCConnector` on the

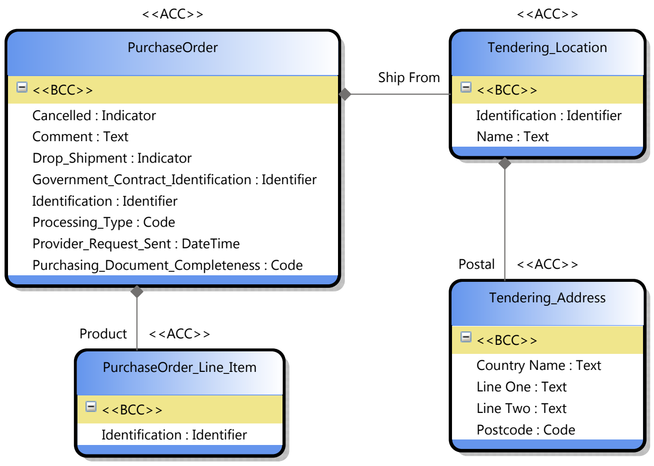

Figure 6: Purchase Order Core Component DSL instance

upper right hand side of Figure 5. The `CCViewDiagram` element represents the diagram, which on the instance level holds the different core components and their relationships. Additionally, the `ACCCompartmentShape` is mapped to the `ACC` domain class. It consists of two decorators (`NameDecorator` and `ACCDecorator`) and one compartment. The name decorator displays the name of the ACC and the ACC decorator simply shows the static text <<ACC>>. This helps to distinguish the different artifacts in the final core component model. The BCC compartment may contain multiple BCCs. For convenience reasons the modeler may optionally choose to expand or collapse the BCC compartment, in order to enhance readability of the core component model.

In the following we dwell on how to use the developed DSL concepts in order to model core component compliant documents. In our example use case for a core component model instance we create a purchase order document. The purchase order must include line items and a ship-from address. Figure 6 shows the example purchase order core component model. Due to space limitations we only show four aggregate core components (ACC) - `PurchaseOrder`, `PurchaseOrder_Line_Item`, `Tendering_Location`, and `Tendering_Address`. Each ACC in this example has at least one basic core component (BCC).

Based on our example we explain a typical core component modeling workflow, and how the DSL representation helps to overcome repetitive modeling tasks. As already outlined in the Section on core components, a business document modeler usually retrieves an existing core component definition from the UN/CEFACT Core Component Library. The generic core component is then tailored to the specific needs of the business domain. Additionally, a business document modeler may also create his own core component definition. However, user created core components are not compatible with UN/CEFACT's core components. Thus, we distinguish between two different tasks: creating a core component definition from scratch and using an existing core component definition.

In order to create a new core component definition from scratch the business document modeler may use the Software Factory for core components, which guides the modeler towards a valid core component definition. Among other components, such as code generators and wizards, the Software Factory for core components also contains the business document modeling environment, which has been generated out of the core component DSL. Using the DSL, the document modeler may drag and drop aggregate core components (ACC) from the toolbox onto the modeling canvas. Additionally, basic core components (BCC) may be added to the ACC and the data types of the basic core components are set. The user is automatically presented with a set of core data types, which he may assign to a basic core component. In case no pertinent core data type (CDT) is present, the user may use the core data type DSL and create a new CDT definition. Using the concept of a connector, an existing ACC may be associated with another ACC in order to represent an association core component (ASCC).

In case an existing core component definition is present in the UN/CEFACT registry, the user may retrieve the core component definition from the registry. Thereby, a well defined import interface, built on top of the business document model designer is used. Utilizing the "Get BIE from CC" wizard, which is part of the Software Factory, a business information entity (BIE) may be derived from the core component definition. Thereby, the user restricts the existing core component definition and is able to automatically generate a business information entity from it. The wizard ensures, that the created business information entity is compliant to the underlying core component. Using the business information entity DSL, the user may assemble different business information entities in order to represent a business document. Eventually, this business document model, based on business information entities, may be used to generate XML Schema artifacts. These XML Schema artifacts may for instance represent message definitions in a service oriented architecture. Having examined the theoretical background, we now explain how the derivation of business information entities from core components is realized on a technical level.

The in-memory representation of DSL models in the DSL Tools is provided by a so called store, which provides the ability to modify models. Hence, the store in the DSL Tools enables the developer to load DSL models and iterate through them or search for specific model elements in it. As shown in Listing 1, each modification in the store has to occur in a transaction (line 1). First, a collection of all aggregate core components is retrieved in line 2. In the next step the collection is iterated and for every ACC an appropriate aggregate business information entity (ABIE) is created and added to a collection (line 8 in Listing 1). Outside of the scope of Listing 1 we additionally have to transform BCCs to BBIEs and ASCCs to ASBIEs. Instead of using the store to generate one model out of another, it could also be used to automatically generate artifacts such as XML Schema out of an model. Although in principle any XML Schema representation may be generated out of a core component model, UN/CEFACT recommends to follow the Naming and Design Rules (NDR) [16], which build on the current Core Component Technical Specification.

Listing 1: Model transformation through store

```
1  using (Transaction transaction = store.TransactionManager.BeginTransaction("
         Generate BIEModel", true)){
2          ReadOnlyCollection<ModelElement> foundaccs = store.ElementDirectory.
                   FindElements(ACC.DomainClassId);
3          if (foundaccs.Count > 0){
4                  curbiev = new BIEModel(mystore);
5                  foreach (ACC acc in foundaccs){
6                          ABIE abie = new ABIE(mystore);
7                          abie.Name = acc.Name;
8                          curbiev.ABIE.Add(abie);
9                  }
10         }
11 }
```

Another approach to derive artifacts is with the help of the Text Template Transformation Toolkit (T4), which is a template-based code generation engine developed by Microsoft. The T4 templates are defined by processing directives, text blocks, and code blocks. Code blocks may be written in C# or Visual Basic .NET.

An important prerequisite for the successful generation of code artifacts from a conceptual model is the validity of the model. Validation of a DSL is either done by specifying soft constraints or hard constraints. Soft constraints are validated at specific tasks such as opening a file, saving a file, or starting the validation through a menu item, and do not provide a correct model constantly throughout the modeling process. For example, a user may model an aggregate core component without any basic core components or association core components, which violates a constraints defined by the Core Component Technical Specification. However, if the user tries to save the model, a validation error message will arise. On the other hand, hard constraints assure the correctness of the model at any time. For example, the user is not able to add a string value in a numeric field at any time.

The DSL tools provide different approaches to enable soft and hard constraints. Soft constraints may either be assigned by adding validation methods to the model class or to the elements class. Hard constraints are directly wired into the DSL definition. There are already some hard constraints embedded within each DSL: maximum multiplicities, type constraints on role players, and type constraints on property values. Nevertheless, the developer is also able to add his own hard constraints. For example, in our core component DSL a hard constraint is used in order to validate, that a basic core component (BCC) references a core data type (CDT), which exists in a CDT model.

Of particular importance for our core component DSL is the possibility to serialize core component definitions in order to allow for storage and retrieval. As outlined at the beginning, a core library holding reusable core component definitions is the central basis for all context specific business information entity definitions. The DSL Tools automatically creates methods for saving and loading DSLs to/from files when compiling the DSL meta model. As a file format XML is used, but the user may also customize the way the data is saved. For example, for each element type it may be defined, if the GUID is saved for it or not. In our solution this is important, since we use the GUIDs to cross-reference between models. For each saved model there are two files created. One for the model data itself and the other one for the appearance of the diagram data. Additionally we created a core component serializer, which allows to save a core component in a registry compliant manner. This was necessary, since the format expected by the core component registry is not the same as the standard XML, generated by the built-in serialization mechanism.

Having specified the most important aspects of our DSL based approach towards a core component compliant implementation, we in the following outline the most important advantages a DSL has compared to a UML based approach.

5 Advantages of the DSL compared to UML

Since its inception in the mid-nineties of the last century, the Unified Modeling Language (UML) has gained considerable attention in particular in the model driven development community. As already outlined in the introduction, UN/CEFACT has also adopted UML in order to transfer the implementation independent core component concepts on an easy to use platform. As a result the UML Profile for Core Components (UPCC) specification has been released. In particular in regard to core component modeling a DSL has a set of advantages, compared to a solution based on UML. Table 1 shows the most important differences between UML and a DSL.

	UML	**DSL**
Dedicated	+/-	+
Extensible	+/-	+
Restrictable	+/-	+
Shape adaptable	-	+
Processable	+/-	+
Holistic	+/-	+

Table 1: UML vs. DSL

Dedicated. DSLs and UML are both used for modeling, but aim at different problem areas. UML may in principle be used to depict any problem scenario in regard to application structure, behavior, architecture as well as business processes, and data structures. In contrast, a DSL leaves out unnecessary aspects and focuses entirely on a specific problem domain e.g. business document modeling. Thus, DSL based solutions are streamlined and less complex than their UML equivalent.

Extensible. The UML is defined on the meta model layer (M2) according to the Meta Object Facility [10]. Consequently, the concepts defined in the UML meta model [11] are used to build UML models on the model layer (M1). In principle anybody may adapt the UML meta-model using the concept of UML profiles in order to customize the generic UML meta-model to a specific application domain. However, UML profiles are still limited in their expressiveness, because they must adhere to the UML meta-model specification. In contrast, a DSL defines its own meta-model and must not consider any predefined limitations. As a result more powerful domain specific solutions are feasible.

Restrictable. As part of its profile mechanism, UML provides the Object Constraint Language (OCL), which allows to restrict the UML meta-model in a formalized manner. However, most of the currently available UML modeling tools cannot interpret OCL. If extension mechanisms of the UML modeling tool are available, OCL interpreters or validators checking the OCL constraints against a UML model must be implemented manually. However, such an approach requires considerable coding effort. In contrast, DSL definitions allow to define customized code, which specifically restricts a DSL. These code fragments, checking the consistency of a DSL based model, are relatively easy to implement. Using the built-in validation feature of the DSL, a user is for instance prevented of nesting core components recursively.

Shape adaptable. In regard to the graphical representation of a UML model, the UML modeler is limited to a well defined set of shapes representing classes, packages. Although the UML meta model would in principle allow to use different shapes e.g. for classes, most UML modeling tools restrict the set of allowed shapes. Although this may be regarded as a benefit, since all modelers have a common understanding of elements by visually recognizing their purpose, domain specific amendments are not possible. In contrast a DSL, using the built-in shape mechanism, allows any visual representation of choice for classes and connections between classes. Thus, any domain specific realization in regard to visual representation of concepts is possible.

Processable. The overall goal of a model-driven approach is the generation of code artifacts (e.g. XML Schema) from a platform independent models (e.g. a conceptual core component model, based on UML). In order to transform a UML based core component model to XML Schema artifacts, the Naming and Design Rules [16] of UN/CEFACT must be reflected. Thus, user specific code generators built on top of UML tools, which are complex and expensive to implement, must be realized. Some UML case tools do not even allow direct access to the tool-internal representation format of the UML model. In contrast a DSL store may be easily accessed using a programming language. With the T4 template mechanism an additional powerful transformation feature is provided.

Holistic. A holistic model-driven development requires a set of different features such as transformation mechanism between different model types, for example platform independent models (PIM) and platform specific models (PSM). Additionally, the transformation of model representations to code representations such as XML Schema must be realized. Although all these features are in principle possible with UML, considerable coding effort and the integration of different tools for the specific tasks is necessary. In contrast, a DSL may be integrated into a Software Factory which comprises code generators, different domain specific languages as well as wizards, guiding a modeler through transformation tasks. Thus, the modeler is provided with a holistic solution approach for a domain specific problem.

6. Related Work

The concept of domain specific development is not new, but has already been introduced in 1976 by Parnas [12] through the concept of program families. The approach included the definition of a program generator that is able to produce program family members. Thus, the general idea of Parnas is comparable to Software Factories as used today. Later in 1985 Bentley [2] discussed the possibility of viewing the work of programmers as the constant invention of *little languages*. His article concluded, that essentially these little languages are designed to solve problems of a particular kind. Thus, Bentley had an early idea of what Domain Specific Languages are used for today.

The concept of domain specific development is also closely related to model-driven development (MDD). In a MDD approach a model is first built using a graphical language such as the Unified Modeling Language (UML) [11]. The models, together with a code generator, are used to transform the conceptual static as well as dynamic system representation into executable code artifacts. One of the most important initiatives in this field is the Model Driven Architecture (MDA) initiative [9] by the Object Management Group (OMG). Nevertheless, a UML based approach has several shortcomings compared to a DSL based approach as we already outlined before.

In particular model driven approaches based on Domain Specific Languages have gained considerable popularity in recent years. Thereby, several approaches aim at streamlining the DSL definitions in order to provide the users with an easy to use DSL definition. Mikkonen et al. [13] present a lightweight model-driven approach using domain specific languages, where a lightweight DSL is used to bootstrap a more heavyweight DSL. Another incremental approach towards defining a DSL is presented by [3].

In particular in a service oriented context model driven approaches may help to overcome changing requirements. A study on different DSL based approaches for service oriented architectures has been conducted by Oberortner et al. [8]. The authors provide a thorough overview of different DSL approaches and evaluate the identified DSLs using prototyping experiments. Another approach to model services based on a domain specific language is presented by Achilleos et al. [1]. The authors realize service creation through the phases of domain specific language definition, model definition and validation, model-to-model transformation, and model-to-code generation. However, their proposed solution does not consider the definition of the exchanged data in a service oriented environment. In this paper we successfully close the gap towards a model driven generation of interface definitions for a SOA.

7. Conclusion and Outlook

In this paper we introduced the basic concepts of the Core Component Technical Specification. We argued, that the implementation neutral core component concepts cannot leverage any benefits for enterprises. Without an appropriate and easy to use implementation technology no additional value is created. In order to close this gap we introduced our approach, based on Domain Specific Languages (DSL). Our DSL based solution, together with the definition of a Software Factory guides a business document modeler towards a valid core component model. The goal is to define core components first on a conceptual and platform independent level. Consequently the platform independent representation may be used to derive XML Schema code artifacts for a Service Oriented Architecture (SOA). Using our model-based approach for the definition of SOA interfaces, changing requirements may be reflected in a more flexible manner. Thereby, our DSL based approach provides several benefits compared to a UML based solution.

Future work aims to overcome some of the pitfalls of a Domain Specific Language. Since there is no open standard for serializing a DSL, conceptual document model interoperability between tools of different vendors is still an open issue. Another open research questions includes the aspect of model versioning. Currently there is no optimistic version control available for a DSL. Accordingly it is quite difficult for two people to work on the same model at the same time and merge the changes in an automated manner.

References

[1] A. Achilleos, K. Yang, and N. Georgalas. A Model Driven Approach to Generate Service Creation Environments. In *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM), November 30 - December 4, New Orleans, LA, USA*, pages 1673–1678, 2008.

[2] J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[3] K. Bierhoff, E. S. Liongosari, and K. S. Swaminathan. Incremental Development of a Domain-Specific Language That Supports Multiple Application Styles. In *Proceedings of the 6th OOPSLA Workshop on Domain Specific Modeling, October 22, Portland, Oregon, USA*, pages 67–78, 2006.

[4] S. Cook, G. Jones, S. Kent, and A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007.

[5] H. Li. XML and Industrial Standards for Electronic Commerce. *Knowledge and Information Systems*, 2(4):487–497, 2000.

[6] P. Liegl. Conceptual Business Document Modeling using UN/CEFACT's Core Components. In *Proceedings Sixth Asia-Pacific Conference on Conceptual Modelling (APCCM), Wellington, New Zealand, January 20-23*, pages 59–69, 2009.

[7] J.-M. Nurmilaakso. EDI, XML and e-Business Frameworks: A Survey. *Comput. Ind.*, 59(4):370–379, 2008.

[8] E. Oberortner, U. Zdun, and S. Dustdar. Domain-Specific Languages for Service-Oriented Architectures: An Explorative Study. In *Proceedings of the First European ServiceWave Conference (ServiceWave), December 10 - 13, Madrid, Spain*, 2008.

[9] Object Management Group (OMG). *Model Driven Architecture*, 2003.

[10] OMG. *MetaObject Facility*. Object Management Group, January 2006.

[11] OMG. *Unified Modeling Language (UML) Infrastructure and Superstructure 2.1.2*, 2007.

[12] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):193–213, 1976.

[13] R. Pitkänen and T. Mikkonen. Lightweight Domain-Specific Modeling and Model-Driven Development. In *Proceedings of the 6th OOPSLA Workshop on Domain Specific Modeling, October 22, Portland, Oregon, USA*, pages 159–168, 2006.

[14] UN/CEFACT. *Core Components Technical Specification 3.0*, 2009.

[15] UN/CEFACT. *UN/CEFACT's Core Component Library UN/CCL*, 2009.

[16] UN/CEFACT. *UN/CEFACT's Naming and Design Rules 3.0*, 2009.