Mapping XML Schema to CCTS

Whitepaper

Christian Eis[1] Fabian Kromer[1] Christian Pichler[1]
Michael Strommer[1]

Copyright © 2009, 2010 Researchstudios Austria Forschungsges.m.b.H

[1]{name}@researchstudio.at

Contents

0.1 To-do . . . 3
0.2 Intent . . . 3
0.3 Abstract . . . 3
0.4 Definitions . . . 3

1 Basic Mapping Concepts 4
1.1 Overview of Possible Mappings . . . 4
1.2 General Notes . . . 4
1.2.1 Mapping Names . . . 4
1.2.2 BIE and MA Naming . . . 4
1.2.3 Mapping Samples . . . 5
1.2.4 Dealing with Multiple Uses of a Complex Type . . . 5
1.3 Explicit Mappings . . . 5
1.3.1 Attribute or Simple element → BCC Mapping . . . 5
1.3.2 Attribute or Simple element → SUP Mapping . . . 7
1.3.3 Complex element → BCC Mapping . . . 8
1.3.4 Complex Element → ASCC Mapping . . . 9
1.4 Implicit Mappings . . . 10
1.4.1 Complex Element → ASMA Mapping . . . 10
1.4.2 Complex type → ACC Mapping . . . 12
1.4.3 Simple Type → CDT Mapping . . . 14
1.4.4 Complex type → CDT Mapping . . . 15
1.4.5 Complex Type → MA Mapping . . . 16
1.5 Generating the Document Root . . . 17
1.5.1 Root Element Mapped to SUP . . . 18
1.5.2 Root Element Mapped to BCC . . . 18
1.5.3 Root Element Mapped to ASCC . . . 19
1.5.4 Root element mapped to ASMA . . . 19
1.5.5 Root MA Name . . . 19
1.6 Generated Libraries . . . 19

2 Advanced Mapping Concepts 20
2.1 XSD Sequence . . . 20
2.1.1 Semantics . . . 20
2.1.2 Problem . . . 20
2.1.3 Solution . . . 21
2.2 XSD Choice . . . 21
2.2.1 Semantics . . . 21
2.2.2 Problem . . . 21
2.2.3 Solution . . . 21

2.3 XSD Any 23
2.3.1 Semantics 23
2.3.2 Problem 24
2.3.3 Solution 24
2.4 XSI Type 24
2.5 XSD All 24
2.5.1 Semantics 24
2.5.2 Problem 25
2.5.3 Solution 25
2.6 XSD Redefine 25
2.6.1 Semantics 25
2.6.2 Problem 25
2.6.3 Solution 25
2.7 XSD Substitution Group 25
2.7.1 Semantics 26
2.7.2 Problem 26
2.7.3 Solution 26
2.8 XSD Annotation/Documentation 26
2.8.1 Semantics 26
2.8.2 Problem 26
2.8.3 Solution 26
2.9 XSD Key/KeyRef 26
2.9.1 Semantics 26
2.9.2 Problem 26
2.9.3 Solution 26
2.10 Semi-Semantic Loss 26
2.10.1 Problem 26
2.10.2 Solution 26
2.11 Split 28
2.11.1 Problem 28
2.11.2 Solution 28
2.12 Merge 30
2.12.1 Problem 30
2.12.2 Solution 30

3 Implementation **32**
3.1 MapForce-based Importer 32
3.1.1 Concept 32
3.1.2 Advantages 32
3.1.3 Issues 32
3.2 Custom-made Mapping Editor 33
3.2.1 Concept 33
3.2.2 User Interface 33
3.2.3 Advantages Over the MapForce-based Implementation 33
3.3 Sub-setting an Imported Document Model 34

0.1 To-do

- Split see 2.11
- Merge see 2.12
- Attributes see **??**
- See if the issue described in Section 2.10 works
- Define mapping restrictions resulting from simple types/CDTs, as well as element/BCC facets, such as cardinality.
- Define mapping restrictions resulting from element/ASCC facets, such as cardinality.
- Define how XSD facets are expressed in the generated UPCC elements.

0.2 Intent

The intent of this document is to explore and define the concepts for mapping and importing XSD-based document standards like ebInterface to UPCC.

0.3 Abstract

The generic importer for XSD-based document standards to UPCC is based on a mapping from the XSD to CCTS concepts. Currently, we define the mapping with Altova MapForce and then import the mapping file, along with the source XSD to generate appropriate UPCC libraries (DOCLibrary, BIELibrary, and BDTLibrary).

This document describes basic mapping concepts, including explicit and implicit mappings, as well as advanced mapping concepts defined by the Altova MapForce mapping. We also describe the expected document structures to be created by the importer. Finally, we discuss implementation alternatives for the mapper/importer.

0.4 Definitions

Simple Element: An XSD element typified through an XSD Simple Type. For example:

```
<xs:element name="simple-element" type="xs:string"/>
```

Complex Element: An XSD element typified through an XSD Complex Type. For example:

```
<xs:element name="complex-element" type="complex-type"/>
<xs:complexType name="complex-type">
  ...
</xs:complexType>
```

Chapter 1

Basic Mapping Concepts

1.1 Overview of Possible Mappings

Table 1.1 gives an overview of the possible mappings between XSD and CCTS concepts. Some of the mappings are explicit (i.e. made by the user) while others are implicit (i.e. automatically derived from the user's explicit mappings). A detailed specification of explicit and implicit mappings can be found in the following sections (see section 1.3 on the next page and section 1.4 on page 10).

		Attribute	**Simple Element**	**Complex Element**	**Simple Type**	**Complex Type**
Explicit	SUP	x	x			
	BCC	x	x	x		
	ASCC			x		
Implicit	CDT				x	x
	ACC					x
	ASMA			x		
	MA					x

Table 1.1: Overview of possible mappings between XSD and CCTS concepts.

1.2 General Notes

1.2.1 Mapping Names

We have chosen core component concept names (SUP, BCC, ASCC, CDT, and ACC) for mappings that map directly to existing core components. Mappings that map to newly created elements have message assembly concept names (ASMA, MA).

1.2.2 BIE and MA Naming

The naming rules for the generated BIEs and MAs have been designed with the following goals in mind:

- Preserve the "based on" dependencies between BBIEs/ASBIEs and BCCs/ASCCs.

- Preserve the original XSD structure as much as possible.
- Ensure that names are unique within their relevant naming scope (e.g. BBIE names must be unique within the containing ABIE; ABIE names must be unique within the BIELibrary).

1.2.3 Mapping Samples

Note that the mapping samples provided in the following sections do not include the root element mappings or the root MA generated for the document (see section 1.5 on page 17).

1.2.4 Dealing with Multiple Uses of a Complex Type

XSD complex types are implicitly mapped to structural elements of CCTS (CDT, ACC, MA). Each complex type can be mapped only once to exactly one CCTS structural element. The importer must be able to deal with cases where a user has mapped a complex type more than once (possibly to different CCTS elements).

1.3 Explicit Mappings

The user can define explicit mappings from source XSD elements and attributes to target CCTS elements. The following mappings are allowed:

- Attribute → BCC mapping
- Simple element → BCC mapping
- Attribute → SUP mapping
- Simple element → SUP mapping
- Complex element → BCC mapping
- Complex element → ASCC mapping

Note that no other explicit mappings are allowed. XSD elements and attributes can be left unmapped. Unmapped attributes and simple elements are ignored, whereas unmapped complex elements are either ignored or implicitly mapped to an ASMA (see section 1.4.1 on page 10).

1.3.1 Attribute or Simple element → BCC Mapping

Mapping Rule 1. An XSD attribute or simple element can be mapped to a BCC.
XSD attributes can only be mapped if

- The complex element containing the XSD attribute is also (explicitly or implicitly) mapped. Otherwise, the attribute mapping is ignored.
- No fixed value is defined in the attribute declaration.
- No default value is defined in the attribute declaration.

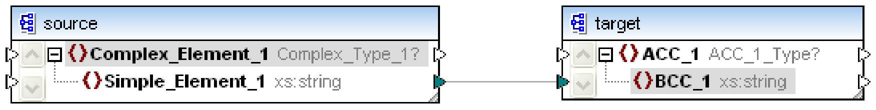


Figure 1.1: A sample Simple element → BCC mapping.

Figure 1.5 on page 8 shows a sample Simple element → BCC mapping where `Simple_Element_1` is explicitly mapped to `BCC_1` of `ACC_1`. The importer is expected to derive the following mappings for this example:

- Complex type → ACC mapping: `Complex_Type_1` → `ACC_1`
  - Simple element → BCC mapping: `Simple_Element_1` → `BCC_1`

Figure 1.6 on page 8 shows the expected document model for the above sample mappings.

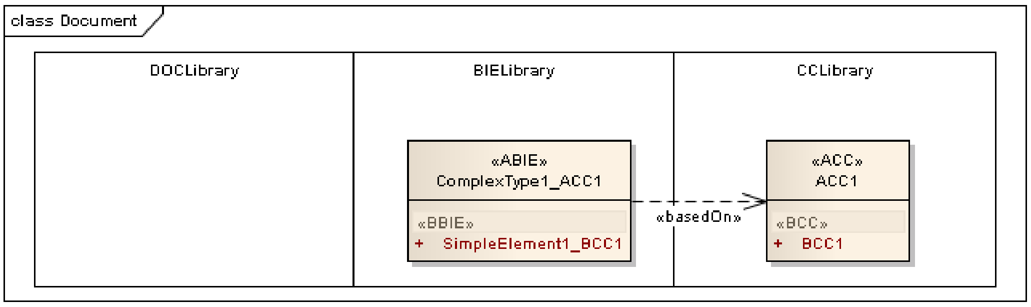


Figure 1.2: Document model for a simple element → BCC mapping.

Implementation Guideline 1. For an Attribute or Simple element → BCC mapping, the importer must generate:

- A BDT in the BDTLibrary following the constraints for Simple Type → BDT mapping as described in section **??** on page ??.
- A BBIE with the following constraints:
  - The BBIE is contained by the ABIE generated for the complex type defining the attribute or simple element.
  - The BBIE is based on the BCC the attribute or simple element is mapped to.
  - The type of the BBIE is the BDT generated for the element's simple type.
  - For attributes, the values of *required* shall be mapped accordingly to the multiplicity constraint of the generated BBIE.
  - For simple elements, generated BBIE names are the BCC names, qualified with the simple element's name.
  - For attributes, generated BBIE names are the BCC names, qualified with the complex type's name and the attribute's name.

1.3.2 Attribute or Simple element → SUP Mapping

Mapping Rule 2. An XSD attribute or simple element used within a complex type, can be mapped to a SUP, if all attributes and simple elements of the complex type are either not mapped or mapped to SUPs of the same CDT.

XSD attributes can only be mapped if

- The complex element containing the XSD attribute is also (implicitly or explicitly) mapped. Otherwise, the attribute mapping is ignored.
- No fixed value is defined in the attribute declaration.
- No default value is defined in the attribute declaration.

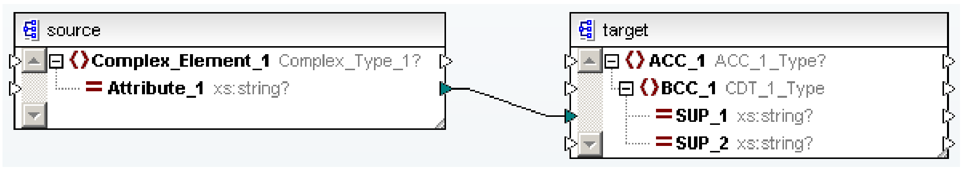


Figure 1.3: A sample Attribute → SUP mapping.

Figure 1.3 shows a sample Attribute → SUP mapping where `Attribute_1` is explicitly mapped to `SUP_1` of `CDT_1`. The importer is expected to derive the following mappings for this example:

- Complex type → CDT mapping: `Complex_Type_1` → `CDT_1`
  - Attribute → SUP mapping: `Attribute_1` → `SUP_1`

Figure 1.4 shows the expected document model for the above sample mappings.

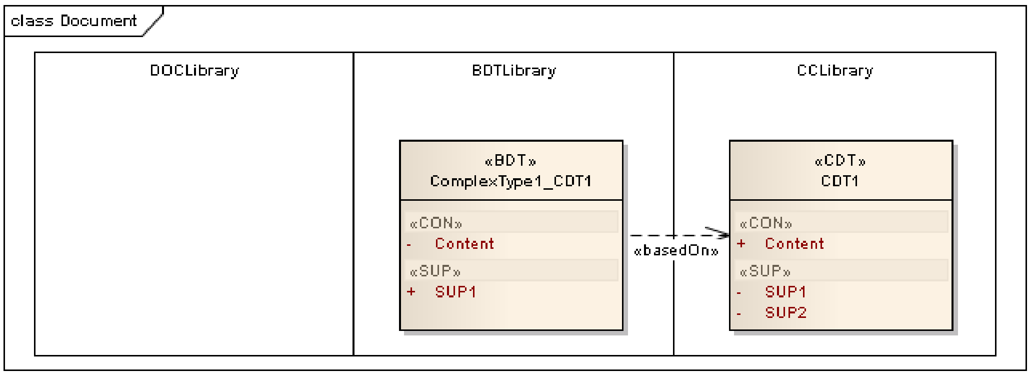


Figure 1.4: Document model for a sample Attribute → BCC mapping.

Implementation Guideline 2. For an Attribute or Simple element → SUP mapping, the importer must generate:

- A SUP with the following constraints:
  - The SUP is contained in the BDT generated for the attribute or simple element's containing complex type.
  - The BDT's SUP must be based on the CDT's SUP that the attribute or simple element is mapped to.
  - The SUP names of the generated BDT must be equal to the SUP names of the CDT that the BDT is based on.

1.3.3 Complex element → BCC Mapping

Mapping Rule 3. An XSD complex element can be mapped to a BCC if the complex type defining the element is mapped to the BCC's CDT.

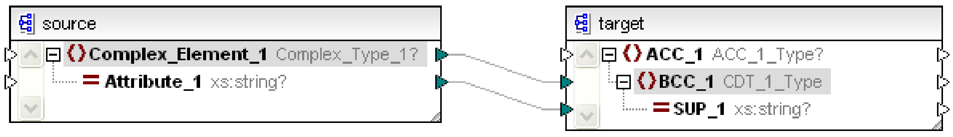


Figure 1.5: A sample Complex element → BCC mapping.

Figure 1.5 shows a sample Complex element → BCC mapping where `Complex_Element_1` is explicitly mapped to `BCC_1` of `ACC_1`. The importer is expected to derive the following mappings for this example:

- Complex element → BCC mapping: `Complex_Element_1` → `BCC_1`
  - Attribute → SUP mapping: `Attribute_1` → `SUP_1`

Figure 1.6 shows the expected document model for the above sample mappings.

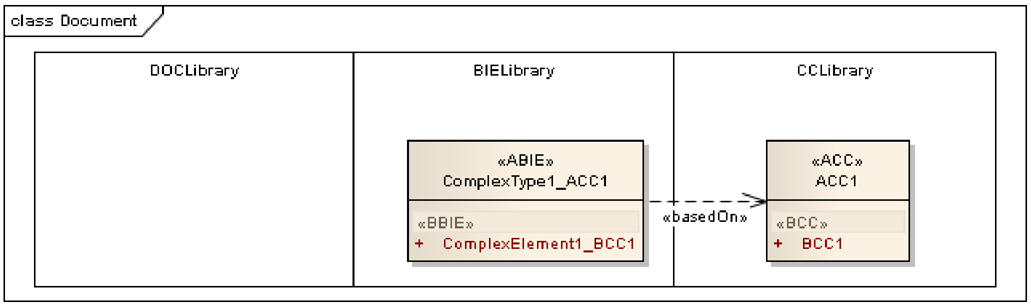


Figure 1.6: Document model for a sample Complex element → BCC mapping.

Implementation Guideline 3. For a Complex element → BCC mapping, the importer must generate:

- A BBIE with the following constraints:
  - The BBIE is contained in the ABIE generated for the complex type containing the complex element.
  - The BBIE is based on the BCC that the complex element has been mapped to.
  - The type of the BBIE is the BDT generated for the complex type of the complex element.
  - Generated BBIE names are the BCC names, qualified with the complex element's name.

1.3.4 Complex Element → ASCC Mapping

Mapping Rule 4. A complex element can be mapped to an ASCC if and only if the complex type of the element is (implicitly) mapped to the ASCC's associated ACC and if the complex element is semantically equivalent to the ASCC (e.g. it is semantically incorrect to map a "home address" to a "work address").

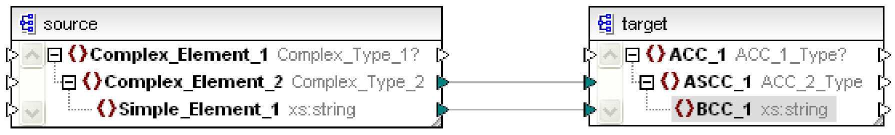

source
Complex_Element_1 Complex_Type_1?
Complex_Element_2 Complex_Type_2
Simple_Element_1 xs:string
target
ACC_1 ACC_1_Type?
ASCC_1 ACC_2_Type
BCC_1 xs:string

Figure 1.7: A sample Complex Element → ASCC mapping.

Figure 1.7 shows a sample ASCC mapping where `Complex_Element_2` is mapped explicitly to `ASCC_1` of `ACC_1`. The mapping is valid because all elements of `Complex_Type_2` are mapped to elements of `ACC_2` (in particular, `Simple_Element_1` is mapped to `BCC_1`). The importer is expected to derive the following mappings for this example:

- Complex Type → ACC mapping: `Complex_Type_1` → `ACC_1`
  - Complex Element → ASCC mapping: `Complex_Element_2` → `ASCC_1`
- Complex Type → ACC mapping: `Complex_Type_2` → `ACC_2`
  - Simple Element → BCC mapping: `Simple_Element_1` → `BCC_1`

Where `ACC_2` is the associated ACC of `ASCC_1`.

Figure 1.8 on the next page shows the expected document model for the above sample mappings.

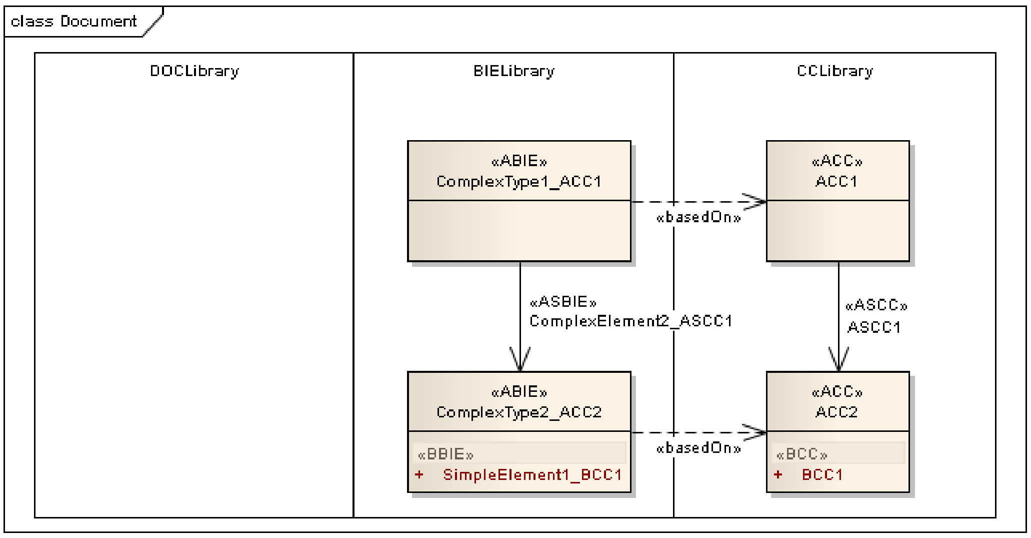


Figure 1.8: Document model for a sample Complex Element → ASCC mapping.

Implementation Guideline 4. For a Complex Element → ASCC mapping, the importer must generate an ASBIE with the following constraints:

- The ASBIE's associating element is the ABIE generated for the complex element's containing complex type.
- The ASBIE's associated element is the ABIE generated for the complex element's own complex type.
- The ASBIE is based on the ASCC the element is mapped to.
- The ASBIE name is the ASCC name, qualified with the complex element name.

1.4 Implicit Mappings

Based on the explicit mappings specified by the user, the importer derives the following implicit mappings:

- Complex element → ASMA mappings
- Complex type → ACC mappings
- Complex type → CDT mappings
- Complex type → MA mappings

1.4.1 Complex Element → ASMA Mapping

Implementation Guideline 5. A complex element is implicitly mapped to an ASMA if and only if

- it is not explicitly mapped, and
- its complex type is (implicitly) mapped to an ACC or MA.

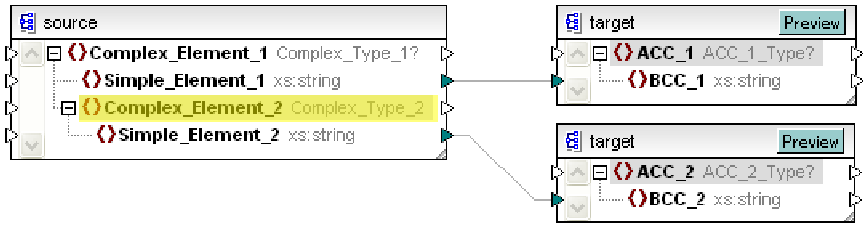

source
Complex_Element_1 Complex_Type_1?
Simple_Element_1 xs:string
Complex_Element_2 Complex_Type_2
Simple_Element_2 xs:string
target
Preview
ACC_1 ACC_1_Type?
BCC_1 xs:string
target
Preview
ACC_2 ACC_2_Type?
BCC_2 xs:string

Figure 1.9: A sample Complex element → ASMA mapping.

Figure 1.9 shows a sample Complex element → ASMA mapping. Actually, the ASMA mapping is not really shown, since it is defined implicitly. `Complex_Element_2` has type `Complex_Type_2`, which is (implicitly) mapped to `ACC_2`, because all of its elements are mapped to elements of `ACC_2`. Therefore, an ASMA mapping for `Complex_Element_2` is implicitly defined, where the associating element of the ASMA will be the MA generated for `Complex_Type_1` and the associated element will be the ABIE generated for `Complex_Type_2`. Note that in this case an explicit ASCC mapping of `Complex_Element_2` is not possible, because `ACC_1` does not contain an appropriate ASCC.

The importer is expected to derive the following mappings for this example:

- Complex Type → MA mapping: `Complex_Type_1` → `[MA_1]`
  - Simple Element → BCC mapping: `Simple_Element_1` → `BCC_1`
  - Complex Element → ASMA mapping: `Complex_Element_2` → `[ASMA_1]`
- Complex Type → ACC mapping: `Complex_Type_2` → `ACC_2`
  - Simple Element → BCC mapping: `Simple_Element_2` → `BCC_2`

Where

- `[MA_1]` is a MA generated in the DOCLibrary for `Complex_Type_1`,
- `[ASMA_1]` is an ASMA associating the ABIE generated for `Complex_Type_2` with `[MA_1]`.

Figure 1.10 on the next page shows the expected document model for the above sample mappings.

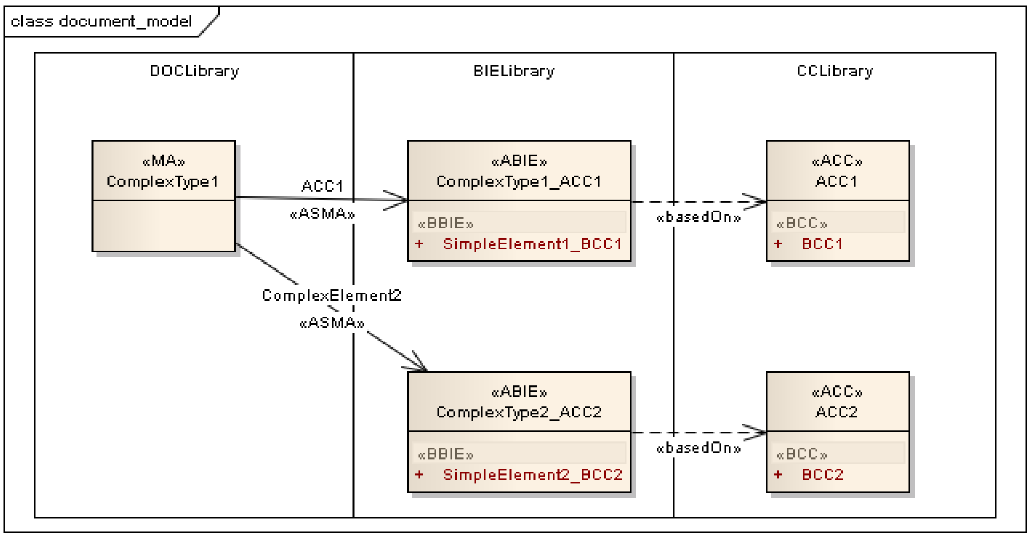


Figure 1.10: Document model for a sample Complex element → ASMA mapping.

Implementation Guideline 6. For a Complex element → ASMA mapping, the importer must generate an ASMA with the following constraints:

- The ASMA's associating element is the MA generated for the complex element's containing complex type.
- The ASMA's associated element is the MA or ABIE generated for the complex element's own complex type.
- The ASMA's name is the complex element's name.

1.4.2 Complex type → ACC Mapping

Implementation Guideline 7. A complex type is implicitly mapped to an ACC if and only if

- all of its attributes and simple elements are either unmapped or mapped to BCCs of that ACC, and
- all of its complex elements are either unmapped or mapped to ASCCs of that ACC.

The samples for BCC and ASCC mappings both are instances of Complex type → ACC mappings. However, figure 1.11 on the preceding page shows a slightly more complex example. All elements of `Complex_Type_1` are mapped to elements of `ACC_1`.

The importer is expected to derive the following mappings for this example:

- Complex type → ACC mapping: `Complex_Type_1` → `ACC_1`
  - Simple Element → BCC mapping: `Simple_Element_1` → `BCC_1`

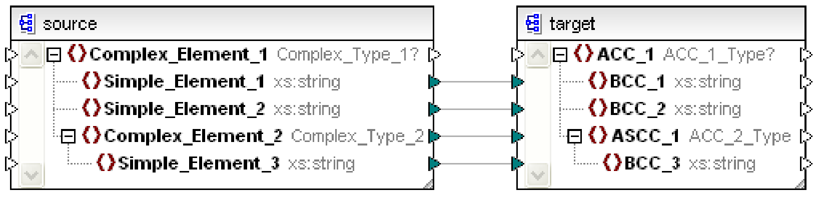

source
Complex_Element_1 Complex_Type_1?
Simple_Element_1 xs:string
Simple_Element_2 xs:string
Complex_Element_2 Complex_Type_2
Simple_Element_3 xs:string
target
ACC_1 ACC_1_Type?
BCC_1 xs:string
BCC_2 xs:string
ASCC_1 ACC_2_Type
BCC_3 xs:string

Figure 1.11: A sample Complex type → ACC mapping.

  - Simple Element → BCC mapping: `Simple_Element_2` → `BCC_2`
  - Complex Element → ASCC mapping: `Complex_Element_2` → `ASCC_1`

- Complex type → ACC mapping: `Complex_Type_2` → `ACC_2`
  - Simple Element → BCC mapping: `Simple_Element_3` → `BCC_3`

Where `ACC_2` is the associated ACC of `ASCC_1`.

Figure 1.12 shows the expected document model for the above sample mappings.

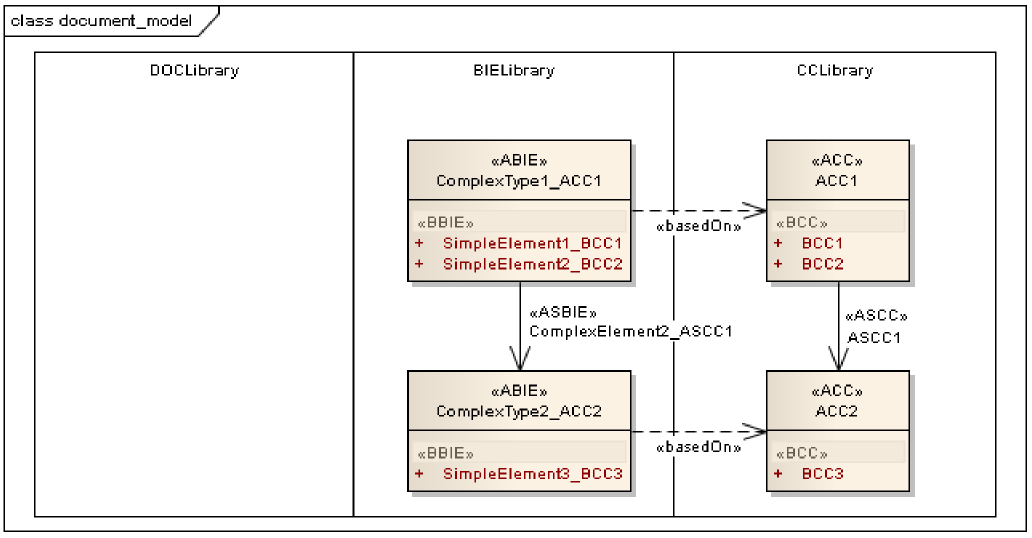


Figure 1.12: Document model for a sample Complex type → ACC mapping.

Implementation Guideline 8. For a Complex type → ACC mapping, the importer must generate an ABIE in the BIE library with the following constraints:

- The ABIE is based on the ACC.
- The ABIE name is the ACC's name qualified with the complex type's name.
- The ABIE must contain a BBIE for each simple element of the complex type with a BCC mapping, following the constraints defined for BCC mappings (see section 1.3.1 on page 5).

- The ABIE must contain an ASBIE for each complex element of the complex type with an ASCC mapping, following the constraints defined for ASCC mappings (see section 1.3.4 on page 9).

1.4.3 Simple Type → CDT Mapping

Implementation Guideline 9. A simple type is implicitly mapped to a CDT if and only if

- at least one simple element utilizing the simple type is mapped to a BCC typified through the CDT.

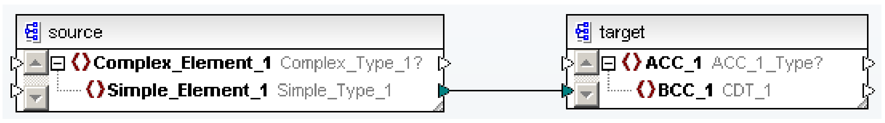

source
Complex_Element_1 Complex_Type_1?
Simple_Element_1 Simple_Type_1
target
ACC_1 ACC_1_Type?
BCC_1 CDT_1

Figure 1.13: A sample Simple Type → CDT mapping.

Figure 1.13 shows an example of a Simple Type → CDT mapping. The simple element `Simple_Element_1` of the complex type `Complex_Type_1` is mapped to `BCC_1` of the ACC `ACC_1`. The importer is expected to derive the following mappings for this example:

- Complex type → ACC mapping: `Complex_Type_1` → `ACC_1`
  - Simple Element → BCC mapping: `Simple_Element_1` → `BCC_1`
- Simple Type → CDT mapping: `Simple_Type_1` → `CDT_1`

Figure 1.14 shows the expected document model for the above sample mappings.

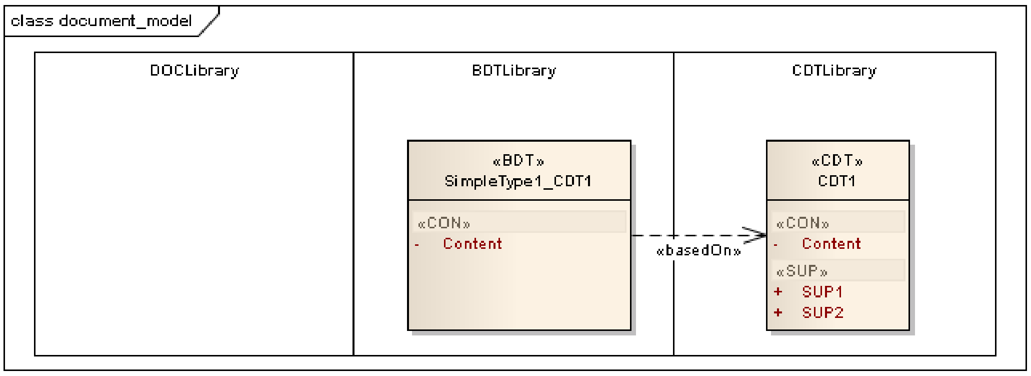


Figure 1.14: Document model for a sample Simple Type → CDT mapping.

Implementation Guideline 10. For an Simple Type → CDT mapping, the importer must generate:

- A BDT in the BDTLibrary with the following constraints:
  - The BDT is based on the CDT.
  - BDTs must be generated only once for each simple type used in the mapping.
  - Generated BDT names are the CDT names, qualified with the simple type name.
  - All facets of the simple type must be applied to the content component of the generated BDT. **TODO:** What facets does XML Schema offer and how can they be mapped to CDTs?

1.4.4 Complex type → CDT Mapping

Implementation Guideline 11. A complex type is implicitly mapped to a CDT if and only if

- all of its attributes and simple elements are either not mapped or mapped to SUPs of that CDT, and
- it does not define any complex elements or none of its complex elements are mapped.

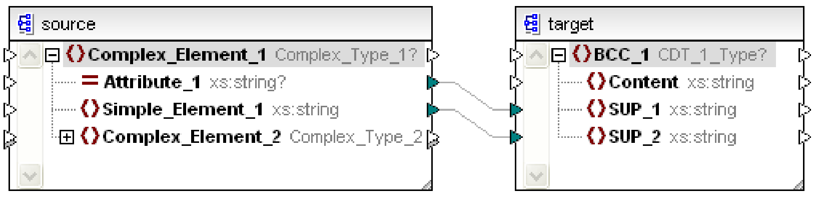


Figure 1.15: A sample Complex type → CDT mapping.

Figure 1.15 shows an example of a Complex type → CDT mapping. All attributes and simple elements of `Complex_Type_1` are mapped to SUPs of `CDT_1`. The importer is expected to derive the following mappings for this example:

- Complex type → CDT mapping: `Complex_Type_1` → `CDT_1`
  - Attribute → SUP mapping: `Attribute_1` → `SUP_1`
  - Simple Element → SUP mapping: `Simple_Element_1` → `SUP_2`

Figure 1.16 on the next page shows the expected document model for the above sample mappings.

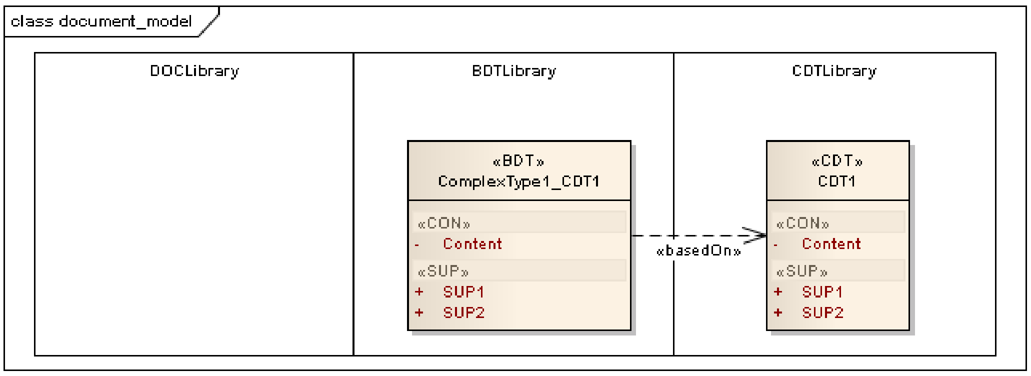


Figure 1.16: Document model for a sample Complex type → CDT mapping.

Implementation Guideline 12. For a Complex type → CDT mapping, the importer must generate a BDT in the BDT library with the following constraints:

- The BDT is based on the CDT.
- The BDT name is the CDT's name, qualified with the complex type's name.
- The BDT must contain a SUP for each attribute or simple element of the complex type with a SUP mapping, following the constraints defined for SUP mappings (see section 1.3.2 on page 7).

1.4.5 Complex Type → MA Mapping

Implementation Guideline 13. A complex type is implicitly mapped to a MA if and only if

- its simple/complex elements are explicitly mapped to BCCs/ASCCs of multiple ACCs and/or
- any of its complex elements are implicitly mapped to ASMAs.

Figure 1.17 shows a sample complex type → MA mapping. The elements of `Complex_Type_1` are mapped to multiple ACCs and `Complex_Element_3` is implicitly mapped to an ASMA. The importer is expected to derive the following mappings for this example:

- Complex Type → MA mapping: `Complex_Type_1` → `[MA_1]`
  - Simple Element → BCC mapping: `Simple_Element_1` → `BCC_1`
  - Simple Element → BCC mapping: `Simple_Element_2` → `BCC_2`
  - Complex Element → ASCC mapping: `Complex_Element_2` → `ASCC_1`
  - Complex Element → ASMA mapping: `Complex_Element_3` → `[ASMA_1]`
- Complex Type → ACC mapping: `Complex_Type_2` → `ACC_3`

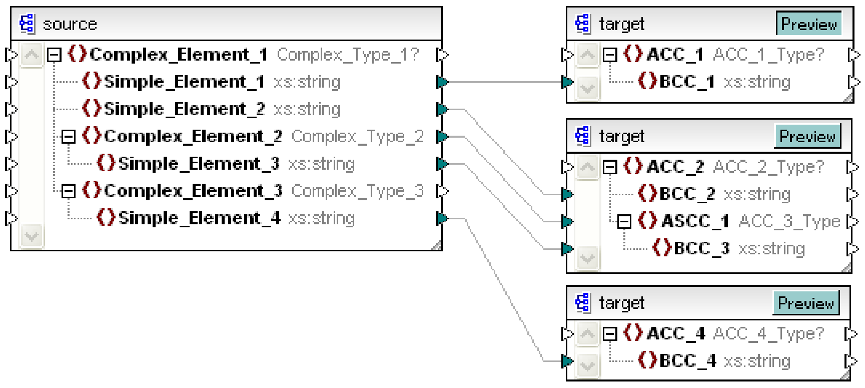

source
Complex_Element_1 Complex_Type_1?
Simple_Element_1 xs:string
Simple_Element_2 xs:string
Complex_Element_2 Complex_Type_2
Simple_Element_3 xs:string
Complex_Element_3 Complex_Type_3
Simple_Element_4 xs:string
target
Preview
ACC_1 ACC_1_Type?
BCC_1 xs:string
target
Preview
ACC_2 ACC_2_Type?
BCC_2 xs:string
ASCC_1 ACC_3_Type
BCC_3 xs:string
target
Preview
ACC_4 ACC_4_Type?
BCC_4 xs:string

Figure 1.17: A sample Complex type → MA mapping.

  - Simple Element → BCC mapping: `Simple_Element_3` → `BCC_3`

- Complex Type → ACC mapping: `Complex_Type_3` → `ACC_4`
  - Simple Element → BCC mapping: `Simple_Element_4` → `BCC_4`

Where

- `ACC_3` is the associated ACC of `ASCC_1`,
- `[MA_1]` is an MA generated in the DOCLibrary for `Complex_Type_1`,
- `[ASMA_1]` is an ASMA associating the ABIE generated for `Complex_Type_3` with `[MA_1]`.

Figure 1.18 on the following page shows the expected document model for the above sample mappings.

Implementation Guideline 14. For a complex type → MA mapping, the importer must generate:

- An ABIE in the BIELibrary for each ACC to which elements of the complex type are mapped with the same constraints as for ACC mappings (applied to those elements that map to the ACC).
- A MA in the DOCLibrary with the following constraints:
  - The MA must contain an ASMA pointing to each of the ABIEs generated in the BIELibrary. The ASMA name is the name of the ACC on which th ABIE is based.
  - The MA must contain an ASMA for each element of the complex type with an ASMA mapping, following the constraints for ASMA mappings (see above).
  - The MA name is the complex type name.

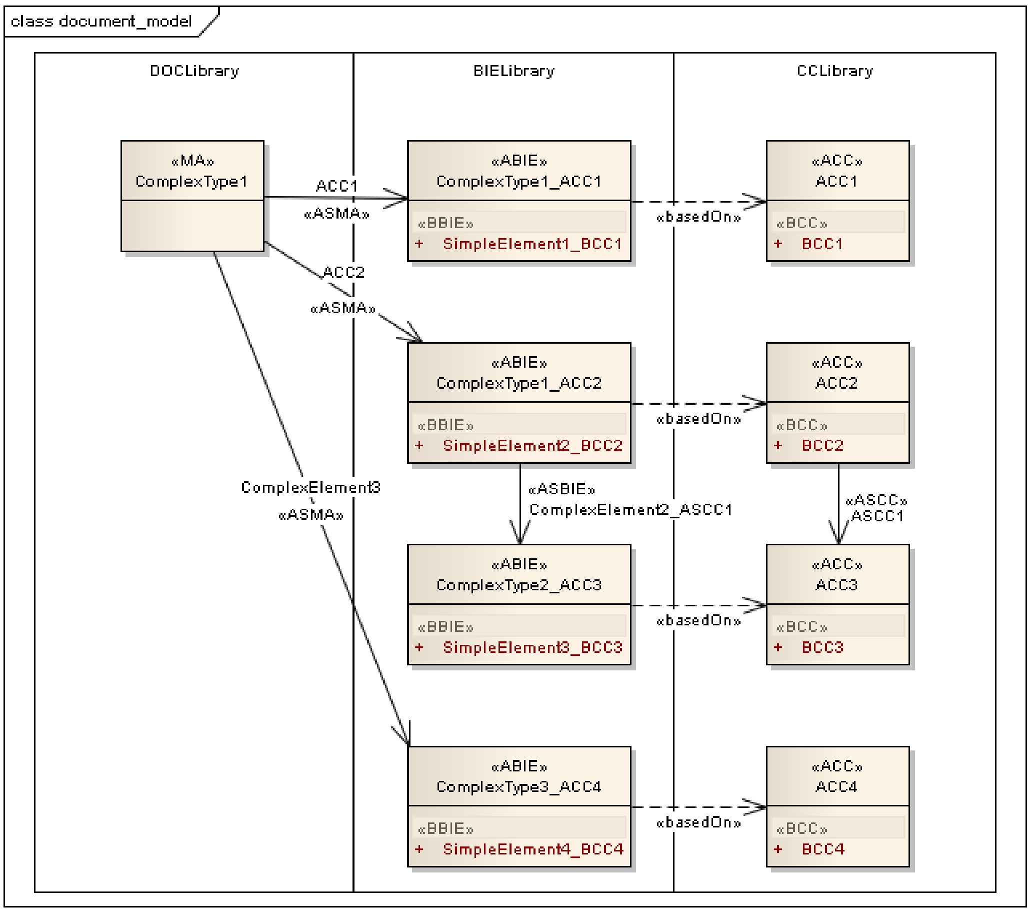


Figure 1.18: Document model for a sample Complex type → MA mapping.

1.5 Generating the Document Root

Each imported document model is represented by a root MA defined in the generated DOCLibrary. The generation pattern for the document root depends on the mapping of the root element of the imported schema. The possible mapping targets of the root element are SUP, BCC, ASCC, and ASMA.

1.5.1 Root Element Mapped to SUP

This is not allowed, because there is no way to generate an ABIE to be aggregated by the root MA, if the only mapping information is a SUP (and its containing CDT).

1.5.2 Root Element Mapped to BCC

This is an extremely rare case and is covered here mostly for completeness, because it implies an XML schema that defines only one global element with a simple type. If the root element is mapped to a BCC, the importer must generate:

- An ABIE in the BIE library, with the following constraints:
  - The ABIE is based on the ACC containing the BCC.

  - The ABIE contains exactly one BBIE following the constraints for BBIEs generated for BCC mappings (see section 1.3.1 on page 5).
  - The ABIE name is the name of the ACC qualified with the domain qualifier.

- A MA in the DOC library, with the following constraints:
  - The MA contains exactly one ASMA where the associated element is the ABIE generated for the ACC containing the BCC.
  - The ASMA name is the name of the associated ABIE.

1.5.3 Root Element Mapped to ASCC

This is not allowed, since mapping the root element to an ASCC does not make any sense.

1.5.4 Root element mapped to ASMA

This is the common case for most XSD mappings. If the root element is (implicitly) mapped to an ASMA, the importer must generate a MA in the DOCLibrary with the following constraints:

- The MA contains exactly one ASMA where
  - The ASMA's associated element is the MA or ABIE generated for the complex element's own complex type
  - The ASMA's name is the complex element's name.

1.5.5 Root MA Name

The name of the document root is a name specified by the user, qualified with the domain qualifier. The default value for the root name might be the name of the schema file. Example: "ebInterface_Invoice", where "ebInterface" is the domain qualifier and "Invoice" is the name of the schema file.

1.6 Generated Libraries

For each document model, we generate the following UPCC libraries:

- A DOCLibrary containing
  - the root MA of the document model, as well as
  - any MAs constructed from Complex Type → MA mappings.
- A BIELibrary containing
  - any ABIEs constructed from Complex Type → ACC mappings, including those where only part of a complex type is mapped to an ACC.
- A BDTLibrary containing
  - any BDTs constructed from Complex Type → CDT mappings, as well as
  - any BDTs constructed for the BCC mappings.

Chapter 2

Advanced Mapping Concepts

The NDR V3.0 specify the following constraints on the construction of XML Schema files:

- xsd:notation MUST NOT be used.
- The xsd:any element MUST NOT be used.
- The xsd:any attribute MUST NOT be used.
- Mixed content MUST NOT be used.
- xsd:redefine MUST NOT be used.
- xsd:substitutionGroup MUST NOT be used.
- xsd:ID/xsd:IDREF MUST NOT be used.

These rules have been specified in order to reach maximum of interoperability. In practice when we deal with arbitrary XML Schema based document standards, we may not assume that these rules hold. In the following sections we consequently deal with XML Schema constructs, which are not covered by UN/CEFACT's NDR.

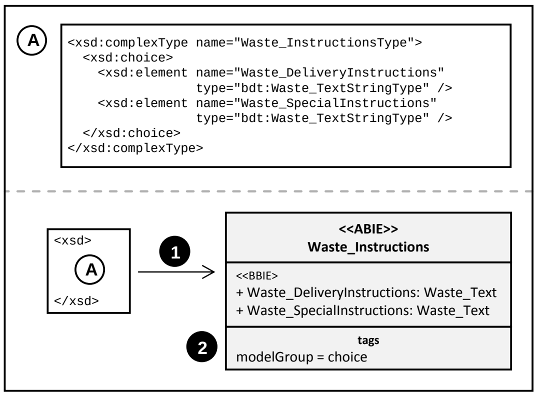
2.1 XSD Sequence

2.1.1 Semantics

A sequence group in XSD is used to determine the order in which the corresponding conforming elements should appear.

2.1.2 Problem

The CCTS standard does not provide a meta property to control for the sequence in which elements should appear. Especially, not for BCCs. However, the UPCC standard does provide a tagged value for BCCs and BBIEs as well as for ASCCs and ASBIEs called *sequencingKey*. This tagged value is meant for implementation purposes such as the serialization to XML Schema.



```
<xsd:complexType name="Waste_InstructionsType">
  <xsd:choice>
    <xsd:element name="Waste_DeliveryInstructions"
                 type="bdt:Waste_TextStringType" />
    <xsd:element name="Waste_SpecialInstructions"
                 type="bdt:Waste_TextStringType" />
  </xsd:choice>
</xsd:complexType>
```

A
<xsd>
A
</xsd>
1
<<ABIE>>
Waste_Instructions
<<BBIE>>
+ Waste_DeliveryInstructions: Waste_Text
+ Waste_SpecialInstructions: Waste_Text
2
tags
modelGroup = choice

Figure 2.1: Reusable Group *choice* (Alternative 1).

2.1.3 Solution

To ensure every element of a complex type is in the right order we have to set the tagged value for the generated BBIE of each corresponding and mapped BCC. Restrictions are:

- In order to maintain consistence across the whole document either all SequencingKey tagged values of BBIEs contained in an ABIE have to be set or none.
- XSD sequence groups have to have a maxOccurs of 1 set.
- XSD sequence groups with maxOccurs greater 1 are ignored.

Open issue: dealing with sequence groups with maxOccurs greater 1.

2.2 XSD Choice

2.2.1 Semantics

The model group *choice* allows the grouping of elements whereas only one of the elements defined may occur. Again, the model group may be used for defining the content of a complex type. An example utilizing the model group is illustrated in Figure 2.1, letter A.

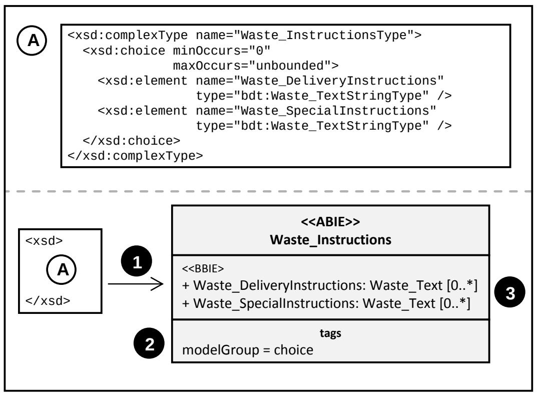
2.2.2 Problem

However, creating an adequate model representation is not feasible without the loss of semantic meaning regarding the characteristics of the model group *choice*. Although the model representation allows specifying minimum and maximum cardinalities, the concept of an exclusive-OR is currently not supported in CCTS.

Another common use case of the model group *choice* is to define element groupings enabling all elements to appear in any desired order as well as in an unlimited number of occurrences. The key to define such a model group is setting the model group's maximum number of occurrences to *unbounded*. An according example is illustrated in Figure 2.2, letter A.

2.2.3 Solution

Overcoming these limitations maybe achieved by extending CCTS the following way. The extension may allow that both elements within the model group are represented as BBIEs



```
<xsd:complexType name="Waste_InstructionsType">
  <xsd:choice minOccurs="0"
             maxOccurs="unbounded">
    <xsd:element name="Waste_DeliveryInstructions"
                 type="bdt:Waste_TextStringType" />
    <xsd:element name="Waste_SpecialInstructions"
                 type="bdt:Waste_TextStringType" />
  </xsd:choice>
</xsd:complexType>
```

A
<xsd>
A
</xsd>
1
2
3
<<ABIE>>
Waste_Instructions
<<BBIE>>
+ Waste_DeliveryInstructions: Waste_Text [0..*]
+ Waste_SpecialInstructions: Waste_Text [0..*]
tags
modelGroup = choice

Figure 2.2: Reusable Group *choice* (Alternative 2).

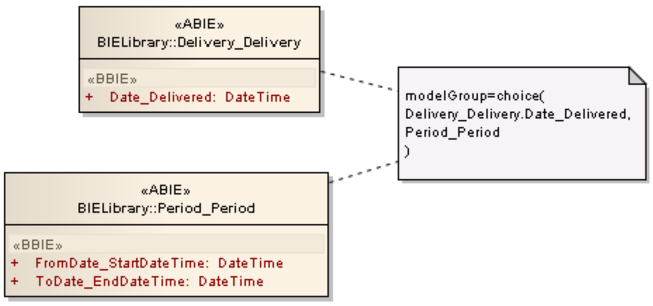

«ABIE»
BIELibrary::Delivery_Delivery
«BBIE»
+ Date_Delivered: DateTime
«ABIE»
BIELibrary::Period_Period
«BBIE»
+ FromDate_StartDateTime: DateTime
+ ToDate_EndDateTime: DateTime
modelGroup=choice(
Delivery_Delivery.Date_Delivered,
Period_Period
)

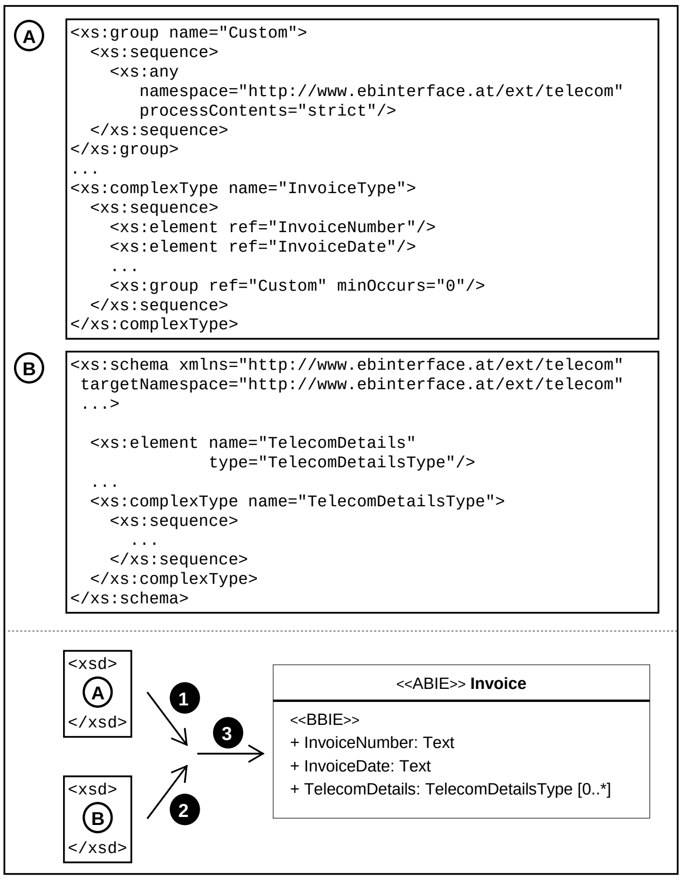
Figure 2.3: Reusable Group *choice* (Alternative 3).

(see Figure 2.1, mark 1). In addition we introduce an additional tagged value named *modelGroup* in ABIEs, indicating that the BBIEs within the ABIE are exclusive (see Figure 2.1, mark 2).

The approach to create an adequate model representation is similar to the approach introduced for the model group *sequence*. All element declarations may be mapped directly to BBIEs (see Figure 2.2, mark 1). The definition of the maximum number of occurrences for the model group itself may be reflected through the cardinalities of BBIEs. Therefore, each BBIE has a minimum cardinality of zero as well as an unlimited maximum cardinality (see Figure 2.2, mark 3).

However, one drawback of the proposed solution is that when serializing the model representation to an XML Schema not all semantics are preserved. According to the naming and design rules an ABIE is serialized into a complex type definition utilizing the model group *sequence* with according minimum and maximum occurrences for each BBIE. Hence, the resulting XML Schema is not the same as the schema used to create the model representation. However, using our introduced tagged value "modelGroup" as illustrated in Figure 2.2, mark 2, the correct semantics are preserved.

In Figure 2.3 we use annotation as a way to mark choice groups, which is less incisive in respect of the modeling language we try to semantically extend.

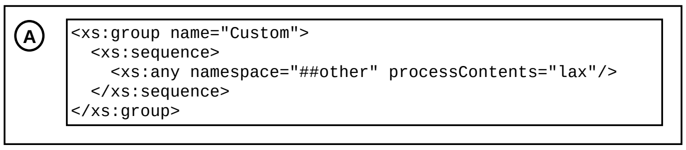


```
<xs:group name="Custom">
  <xs:sequence>
    <xs:any
       namespace="http://www.ebinterface.at/ext/telecom"
       processContents="strict"/>
  </xs:sequence>
</xs:group>
...
<xs:complexType name="InvoiceType">
  <xs:sequence>
    <xs:element ref="InvoiceNumber"/>
    <xs:element ref="InvoiceDate"/>
    ...
    <xs:group ref="Custom" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

```
<xs:schema xmlns="http://www.ebinterface.at/ext/telecom"
 targetNamespace="http://www.ebinterface.at/ext/telecom"
 ...>

  <xs:element name="TelecomDetails"
              type="TelecomDetailsType"/>
  ...
  <xs:complexType name="TelecomDetailsType">
    <xs:sequence>
      ...
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Figure 2.4: Element *xsi:any*.



```
<xs:group name="Custom">
  <xs:sequence>
    <xs:any namespace="##other" processContents="lax"/>
  </xs:sequence>
</xs:group>
```

Figure 2.5: Excerpt from the ebInterface standard.

2.3 XSD Any

2.3.1 Semantics

The `xs:any` construct in XML Schema is used for declaring placeholders in an XML schema which in turn adds greater flexibility in regard to the structure of the XML schema. As a consequence, stakeholders representing information according to an XML schema, which utilizes the `xs:any` construct, may encode any content at the location where the construct was used (see Figure 2.4, letter A). The location where custom content may be stored is from now on referred to as custom section. The only condition that the content in the custom section must fulfill is that it is well-formed. Furthermore, the construct allows the use of certain attributes that may be used to further restrict the content in the custom section. In particular, these attributes allow restrictions in regards of the namespace that the content must belong to as well as in regards of an XML schema that the structure of the content must comply with.

2.3.2 Problem

Therefore, for creating an adequate model representation, it is necessary to address the namespace restrictions as well as the content restrictions. For reasons of simplicity we don't address namespace restrictions but focus on content restrictions in more detail. Restrictions on the content are achieved through the `processContents` attribute of the `xs:any` construct. The attribute may have one out of the following three values: `skip`, `lax`, and `strict`. The attribute value `skip` specifies that the content stored in the custom section may be any content that is well-formed. Furthermore, `lax` specifies that if an XML schema describing the content is available then the content may be validated against the particular schema. At last, the attribute value `strict` specifies that an XML schema must be available which describes the content stored in the custom section.

2.3.3 Solution

To summarize, the only case where the structure in the custom section is foreseeable is the third case where the attribute `processContents` is set to `strict`. Therefore, the third case is also the only case where an adequate mapping strategy of the `xs:any` construct to core components is feasible. The mapping strategy in particular means that based on the XML schema utilizing the `xs:any` construct (see Figure 2.4, Letter A) and the XML schema providing an XML schema for the content stored in the custom section (see Figure 2.4, Letter B) it is possible to create a model representation.

Figure 2.5, Letter A, shows an excerpt from the ebInterface standard where the xs:any construct is used. However, an adequate model representation for the `xs:any` construct can not be created since no XML schema for the structure of the custom section is provided nor enforced through the attribute `processContents`. Therefore, it is not feasible to create an adequate model representation for the `xs:any` construct used in the ebInterface standard.

Possible alternative solution: `xs:any` might be mapped to ACC `Note` if it is set to `skip` or `lax`. The enclosing complex type would then be implicitly mapped to a MA. This solution would preserve the semantics that the content might be anything.

2.4 XSI Type

Is not going to be handled. See evaluation papers on this topic.

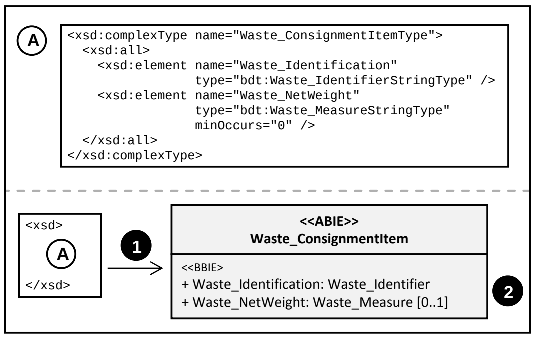
Possible solution: The sub-types of the specified type might be mapped to ACCs or MAs and the resulting ASBIEs might be annotated in a fashion similar to our solution for `xs:choice`.

2.5 XSD All

This concept does not occur within ebInterface and thus has no priority.

2.5.1 Semantics

The third kind of model group named *all* allows to define that all element declarations within the model group may appear in any order, but each element may occur only once. An example is shown in Figure 2.6, letter A.



```
<xsd:complexType name="Waste_ConsignmentItemType">
  <xsd:all>
    <xsd:element name="Waste_Identification"
                 type="bdt:Waste_IdentifierStringType" />
    <xsd:element name="Waste_NetWeight"
                 type="bdt:Waste_MeasureStringType"
                 minOccurs="0" />
  </xsd:all>
</xsd:complexType>
```

A
<xsd>
A
</xsd>
1
<<ABIE>>
Waste_ConsignmentItem
<<BBIE>
+ Waste_Identification: Waste_Identifier
+ Waste_NetWeight: Waste_Measure [0..1]
2

Figure 2.6: Reusable Group *all*.

2.5.2 Problem

UN/CEFACT's NDR do not provide a mapping from XSD All Groups to concepts defined in CCTS. Also the grouping mechanisms provided for BCCs and BBIEs is not sufficient to fully map this XML Schema concept.

2.5.3 Solution

The process for creating an adequate model representation of the model group is proposed the following way. First of all each element declaration within the model group may be mapped directly to a BBIE (see Figure 2.6, mark 1). Furthermore, the minimum occurrences and maximum occurrences of the element declarations may be reflected through the cardinalities of the BBIEs (see Figure 2.6, mark 2). Applied to the example above this would mean that the element *Waste_Identification* is represented through a BBIE with a minimum cardinality of one and the element *Waste_NetWeight* is represented through a BBIE as well whereas the minimum cardinality of the BBIE *Waste_NetWeight* equals zero.

2.6 XSD Redefine

This concept does not occur within ebInterface and thus has no priority.
TODO

2.6.1 Semantics

2.6.2 Problem

2.6.3 Solution

2.7 XSD Substitution Group

This concept does not occur within ebInterface and thus has no priority.
TODO

2.7.1 Semantics

2.7.2 Problem

2.7.3 Solution

2.8 XSD Annotation/Documentation

This concept does not occur within ebInterface and thus has no priority.
TODO

2.8.1 Semantics

2.8.2 Problem

2.8.3 Solution

2.9 XSD Key/KeyRef

This concept does not occur within ebInterface and thus has no priority.
TODO

2.9.1 Semantics

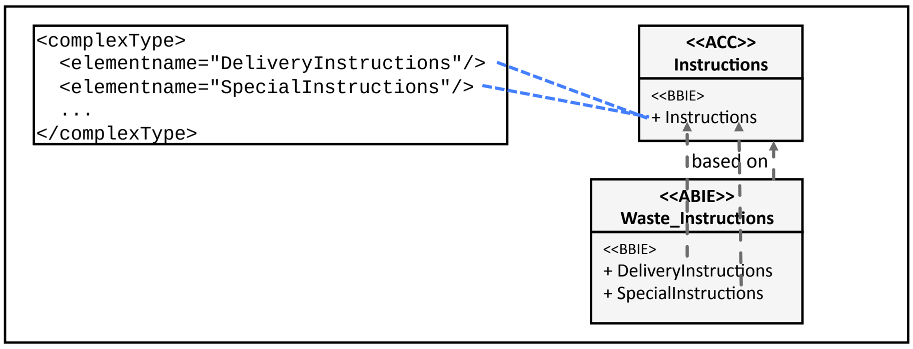
2.9.2 Problem

2.9.3 Solution

2.10 Semi-Semantic Loss

2.10.1 Problem

CCL often expresses concepts in a very generic way. In other languages, however, there exist specializations of these concepts.



```
<complexType>
  <elementname="DeliveryInstructions"/>
  <elementname="SpecialInstructions"/>
  ...
</complexType>
```

Figure 2.7: Conceptual mapping.

2.10.2 Solution

Some BCCs in the CCL are so generic that they may be used for more than one contextualized BBIE. Multiple identifiers of some sort in an invoice document are a good example, where the reuse of one BCC makes sense. The distinction of the BBIEs based on just

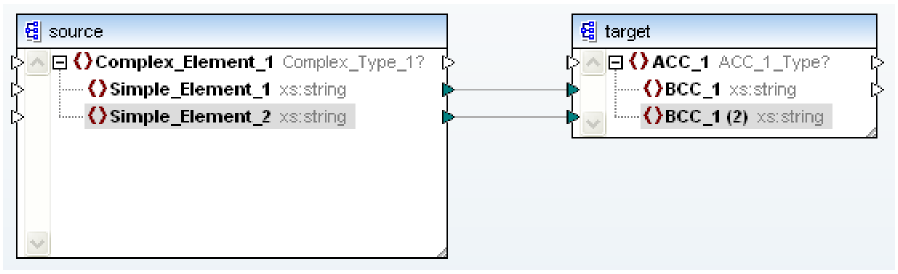

source
Complex_Element_1 Complex_Type_1?
Simple_Element_1 xs:string
Simple_Element_2 xs:string
target
ACC_1 ACC_1_Type?
BCC_1 xs:string
BCC_1 (2) xs:string

Figure 2.8: Example 1: Mapping of two different xsd elements to the same BCC.

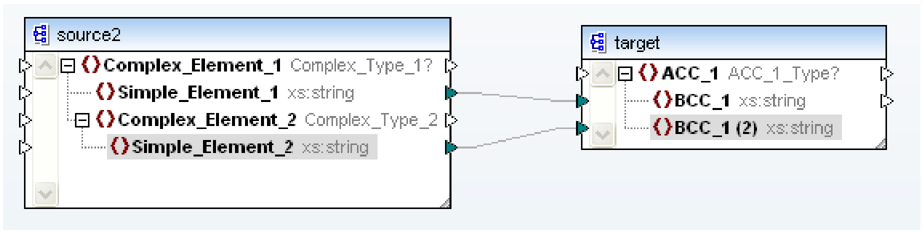

source2
Complex_Element_1 Complex_Type_1?
Simple_Element_1 xs:string
Complex_Element_2 Complex_Type_2
Simple_Element_2 xs:string
target
ACC_1 ACC_1_Type?
BCC_1 xs:string
BCC_1 (2) xs:string

Figure 2.9: Example 2: Mapping of two different xsd elements to the same BCC.

one BCC is achieved through qualifiers stating the specific business context. By using this BCC overloading we are able to map two different elements on one BCC that evolves into two BBIEs on the business information entity layer. Thus, the basic semantic of each BBIE is defined through the based on dependency to the underlying core component, but the semantic of the specifics of that element is not defined. Hence, it is lost during the mapping and modeling task. An example in the waste management context is given in Figure 2.7

Mapping Procedure Figures 2.8 and 2.9 show examples how the mapping described above should be realized within the MapForce environment between some document format and core components.

The following properties apply:

- A generic BCC exists.
- At least two XSD elements or XSD attributes map to this BCC in a general way.
- The first mapping maps to the BCC itself.
- For the second mapping and the ones thereafter a duplicate of the BCC must be created to get a valid MapForce mapping.
- The upper bound multiplicity of the BCC must be at least 2. In case of 0..1 or 1..1 multiplicity the mapping in terms of duplication is not allowed.
- For the original BCC and its duplicates BBIEs are generated. The naming convention from the BIE generation pattern ensures that elements may be distinguished properly.
- The XSD elements or XSD attributes may reside in one complex type (c.f. Figure 2.8).

- The XSD elements or XSD attributes may also reside in different complex types (c.f. Figure 2.9). Structure is preserved through message assembly mechanism described above for the DocLibrary.

2.11 Split

2.11.1 Problem

One simple element comprising two or more concepts shall be mapped upon corresponding CCL concepts. Case analysis shows at least 4 different strategies may be taken:

Case 1 Comprising simple element ↔ single comprising BCC (c.f. Figure 2.10).

Case 2 Comprising simple element ↔ two or more separate BCCs (c.f. Figure 2.11).

Case 3 Comprising simple element ↔ a limited number of BCCs (c.f. Figure 2.12).

Case 4 Comprising simple element ↔ BCCs and duplicates of BCCs(c.f. Figure 2.13).

2.11.2 Solution

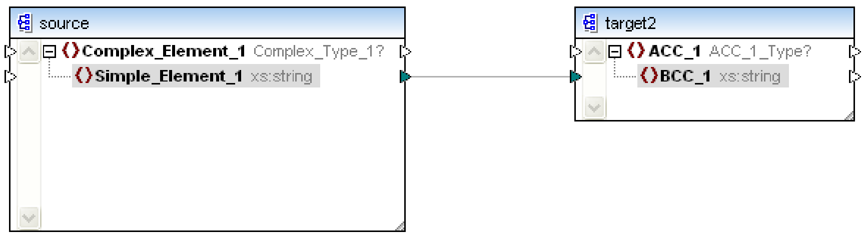


Figure 2.10: Splitting by example. Case 1

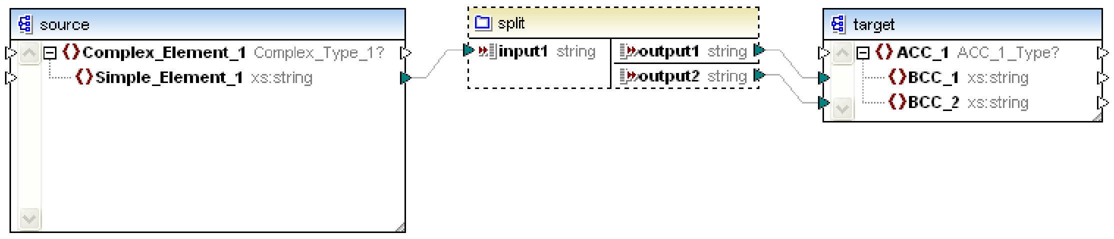


Figure 2.11: Splitting by example. Case 2.

Mapping Procedure Case 1 is the straightforward case that needs no special handling. For the general split function, i.e, Case 2, we define the mapping preconditions and process as follows:

- A single element incorporating at least two different concepts is present in an arbitrary XSD Schema based document model, c.f. Figure 2.11 Simple_Element_1.

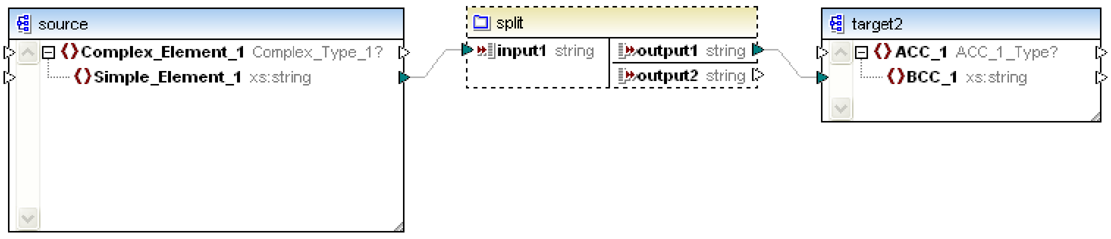


Figure 2.12: Splitting by example. Case 3

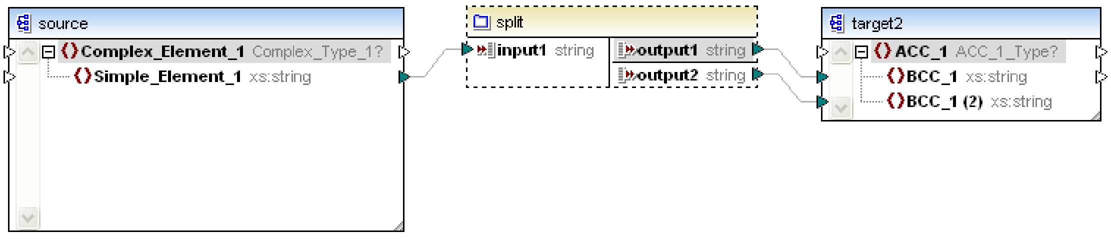


Figure 2.13: Splitting by example. Case 4

- In the CCL separate elements exist for each of the concepts, e.g. BCC_1 and BCC_2 in Figure 2.11.
- Define a constant for the split function that holds a token or pattern on how to perform a merge.
- Insert a custom split function into the mapping file.
- Connect the one simple element to the one input pin.
- Connect two or more output pins to the corresponding BCC elements.

else we define Case 4:

- A single element incorporating at least two different concepts is present in an arbitrary XML Schema based document model, c.f. Figure 2.11 Simple_Element_1.
- In the CCL no separate elements exist for each of the concepts, just BCC_1 in Figure 2.13.
- Duplicate the concept of a BCC in order to store the concept on the other side.
- Qualify the BCCs properly in order to distinguish duplicates, e.g., use output pin name of the split function or use numbers.
- Insert a custom split function into the mapping file.
- Connect the one simple element to the one input pin.
- Connect two or more output pins to the corresponding BCC elements.

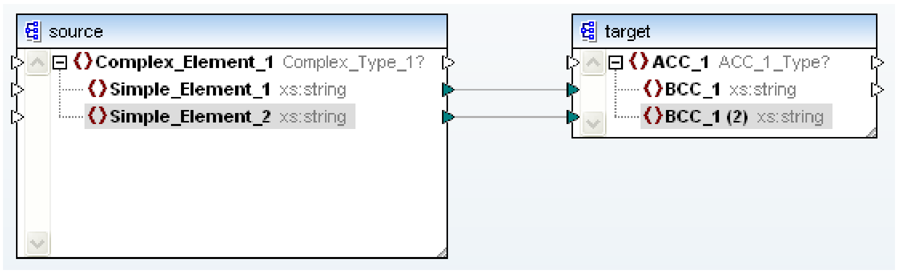
Case 3 is a specialization of Case 2 and will not be dwelled on any further.

2.12 Merge

2.12.1 Problem

Two or more concepts shall be mapped upon a single BCC. No other BCC covering these concepts is available.

2.12.2 Solution



source
Complex_Element_1 Complex_Type_1?
Simple_Element_1 xs:string
Simple_Element_2 xs:string
target
ACC_1 ACC_1_Type?
BCC_1 xs:string
BCC_1 (2) xs:string

Figure 2.14: Example 1: No Merge at all.

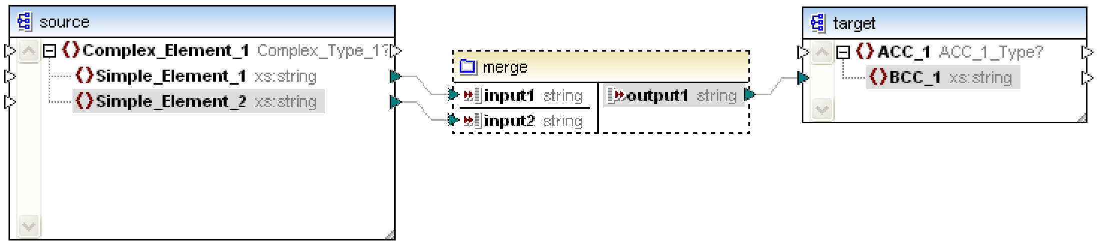


Figure 2.15: Example 2: Merge with explicit mapping function.

Mapping Procedure In most cases when dealing with CCL and other document languages we may apply the following simple mapping strategy:

- Two different concepts in an XML Schema based document language shall be mapped to a single concept in CCL.
- This conceptually means a merge.
- Map the first simple element to the in general corresponding BCC (c.f. Figure 2.14 Simple_Element_1 ↔ BCC_1).
- Map the second simple element to a duplicate of the BCC (c.f. Figure 2.14 Simple_Element_2 ↔ BCC_1(2)).
- Make sure that the multiplicity constraint of the corresponding BCC has a upper bound greater than 1.

If the upper bound of the BCC is 1 the following steps have to be performed:

- Insert a merge function by drag and drop to the mapping model.

- Connect at least two simple elements to the input pins of the merge function (c.f. Figure 2.15 Simple_Element_1 $\leftrightarrow$ input1 and Simple_Element_2 $\leftrightarrow$ input2).
- Connect one output pin to exactly one BCC (c.f. Figure 2.15 output1 $\leftrightarrow$ BCC_1).
- Define a constant for the merge function that holds a token or pattern on how to perform a split.
- Make sure that the multiplicity constraint of the corresponding BCC has a upper bound of exactly 1.
- The number of input elements in the merge function is 2..*.
- The number of output elements in the merge function is 1.

Chapter 3

Implementation

3.1 MapForce-based Importer

3.1.1 Concept

MapForce from Altova Inc. provides graphical tooling for mapping one or many source XML Schemas to one or many target XML Schemas specified by the user. Each element on the source side may thereby be mapped via an output pin to a corresponding element on the target side via an input pin. The mapping from one element to the other is visualized through lines connecting similar concepts. To enhance the semantic meaning of those mappings the user may choose from a large list of predefined mapping functions or specify her own.

3.1.2 Advantages

- Offers stable and proof user interface.
- Allows the definition of custom mapping functions and mapping constants.
- Supports the partition of the mapping process into smaller and easier to handle subtasks.
- XSLT transformations for XML instances may be generated automatically from mapping models.

3.1.3 Issues

There are a number of issues with this implementation and especially with the use of MapForce as a mapping tool. Most of these issues stem from the fact that MapForce is a general purpose mapping tool with lots of complex functionality that we do not need, but lacking certain functionality that would be useful in our special case.

- Some information from the XSD is hidden in MapForce. MapForce only shows information about the structure of the source XSD, but does not show details about the individual elements and types. For example, it does not show whether elements are grouped in a sequence or in a choice.
- There is no way in MapForce to prevent the user from creating invalid mappings. For example, we cannot prevent a user from mapping a string element to a date element. Although I guess that there should be some sort of type checking in MapForce that

might be exploited for this simple case, this remedy does not extend to more specific cases, such as the user mapping a simple-typed element to an ASCC.

- Since there is no API for processing MapForce mappings, our implementation is quite fragile, as it depends on our own (quite limited) knowledge about the mapping files.

3.2 Custom-made Mapping Editor

3.2.1 Concept

As an alternative to using MapForce, we might implement our own mapping editor as part of the VIENNA-Add-In.

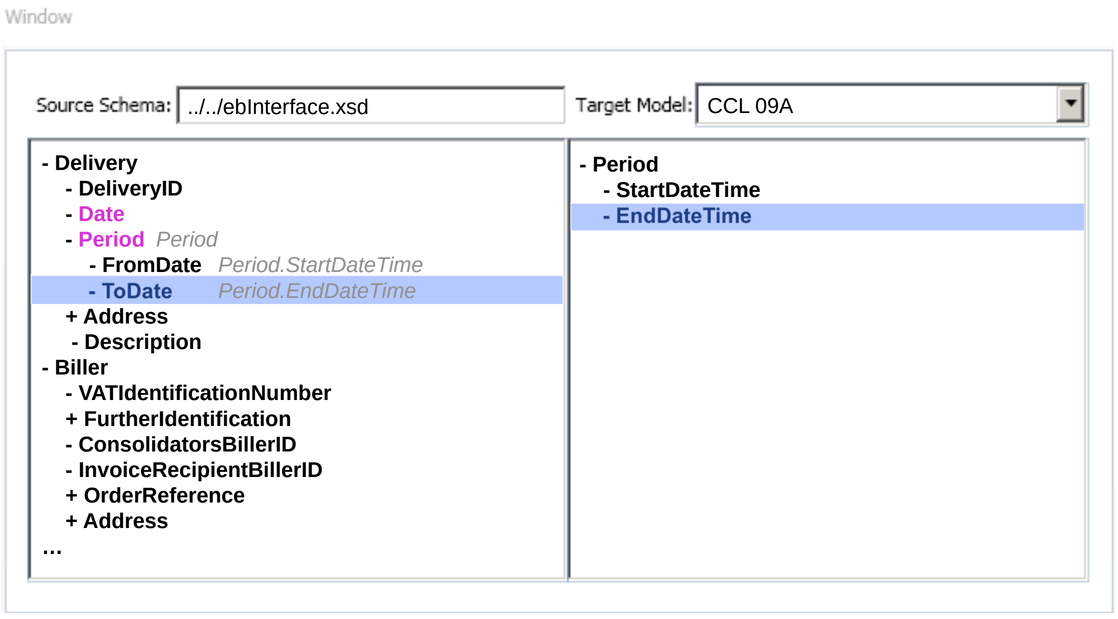
3.2.2 User Interface

The user interface could be quite simple

1. The user opens a wizard, e.g. via a menu item "Import XSD with custom mapping".
2. The user enters either
   (a) an XSD file and a root element to map or
   (b) a previously saved mapping file.
3. The wizard displays a mapping editor (c.f. Figure 3.1), which is a combination of a tree view and a table. It contains the imported document structure as a tree and provides available mapping options for each element. For example, a simple element can be mapped to a BCC, so the wizard adds a "map to BCC" button to the row. If the element is already mapped, the target element is displayed.Implicit mapping for complex types are also displayed. Mapping options can be restricted accordingly, so that the mapping process is efficient. For example:
   (a) Only compatible BCCs are presented as possible targets for a simple element.
   (b) If a simple element of a complex type has already been mapped to a BCC, present the BCCs of the corresponding ACC first, when mapping another simple element of the same complex type.
4. The user can then either
   (a) save the mapping or
   (b) generate the document model.

3.2.3 Advantages Over the MapForce-based Implementation

- User mistakes can be prevented by presenting only valid mapping options.
- The mapping process can be implemented very efficiently (from the user's point of view), because it is tailored to our specific needs.
- The document structure is mapped directly to CCL items, instead of an XSD representation of the CCL.
- We can provide more detailed (and relevant) information to the user, regarding element types and XSD structures.



Source Schema: ../../ebInterface.xsd
Target Model: CCL 09A

- Delivery
  - DeliveryID
  - Date
  - Period Period
    - FromDate Period.StartDateTime
    - ToDate Period.EndDateTime
  + Address
  - Description
- Biller
  - VATIdentificationNumber
  + FurtherIdentification
  - ConsolidatorsBillerID
  - InvoiceRecipientBillerID
  + OrderReference
  + Address
...

- Period
  - StartDateTime
  - EndDateTime

Figure 3.1: Basic mockup for custom mapping editor.

3.3 Sub-setting an Imported Document Model

One of the final usage goals for importing document models is the ability to sub-set an imported document model and then export an XSD defining a sub-set of the originally imported model. An open question is how the sub-setting will actually be done in our tool. Actually there seems to be only one correct solution (with respect to CCTS), based on qualifier hierarchies: While every ABIE must be based on an ACC, it is possible to define a tree-like structure of ABIEs, where lower-level ABIEs are restrictions of higher-level ABIEs. This is called a "qualifier hierarchy" in the CCTS Specification: Multiple qualifiers create a qualifier hierarchy, with each additional qualifier reflecting a further restriction to its less qualified BIE property. Based on this concept, sub-setting can be implemented as follows:

1. Import the ebInterface XSD.

2. Create a clone of the imported document structure (using a wizard), which has an additional qualifier (e.g. "my_ebInterface_Address").

We also need to implement validation routines (e.g. as part of the on-the-fly validation) that check whether qualified ABIEs actually restrict their parents in the qualifier hierarchy.

Bibliography