Content Assembly Mechanism (CAM) Specification Document

Committee Draft V1.0, March 2004

CHANGE HISTORY

Status	Version	Revision	Date	Editor	Summary of Changes
Draft	1.0	0.10	30 December, 2002	DRRW	Rough Draft
		0.11	12th February, 2003	DRRW	Initial Draft
		0.12	23rd February, 2003	DRRW	Revision for comments to 28/02/2003
		0.13	17th May, 2003	DRRW	Revision for comments to 08/05/2003
		0.14	13th August, 2003	DRRW	Revision for comments to 15/08/2003
		0.15	3rd February, 2004	DRRW	Final edits prior to first public release
		0.16	15th February, 2004	DRRW	Release Candidate for Committee Draft CAM
		0.17	19th February 2004	MMER	Edited detailed comments into draft.
Committee Draft		0.17C	12th March 2004	DRRW	Cosmetic changes to look of document to match new OASIS template and notices statement.

DOCUMENT RIGHTS STATEMENT

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS's procedures with respect to rights in OASIS specifications can be found at the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification, can be obtained from the OASIS Executive Director.

OASIS invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to implement this specification. Please address the information to the OASIS Executive Director.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE."

Where, pursuant to a notification under this Policy, the OASIS Board of Directors is aware at the time of publication of proprietary rights claimed with respect to an OASIS specification, or the technology described or referenced therein, such specification shall contain the following notice:

OASIS has been notified of intellectual property rights claimed in regard to some or all of the contents of this specification. For more information consult the online list of claimed rights. This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

TABLE OF CONTENTS

1 Acknowledgements

OASIS wishes to acknowledge the contributions of the members of the CAM Technical Committee to this standards work.

2 Introduction

The Content Assembly Mechanism (CAM) provides an open XML based system for using business rules to define, validate and compose specific business documents from generalized schema elements and structures.

A CAM rule set and document assembly template defines the specific business context, content requirement, and transactional function of a document. A CAM template must be capable of consistently reproducing documents that can successfully carry out the specific transactional function that they were designed for. CAM also provides the foundation for creating industry libraries and dictionaries of schema elements and business document structures to support business process needs.

The core role of the OASIS CAM specifications is therefore to provide a generic standalone *content assembly mechanism* that extends beyond the basic structural definition features in XML and schema to provide a comprehensive system with which to define dynamic e-business interoperability.

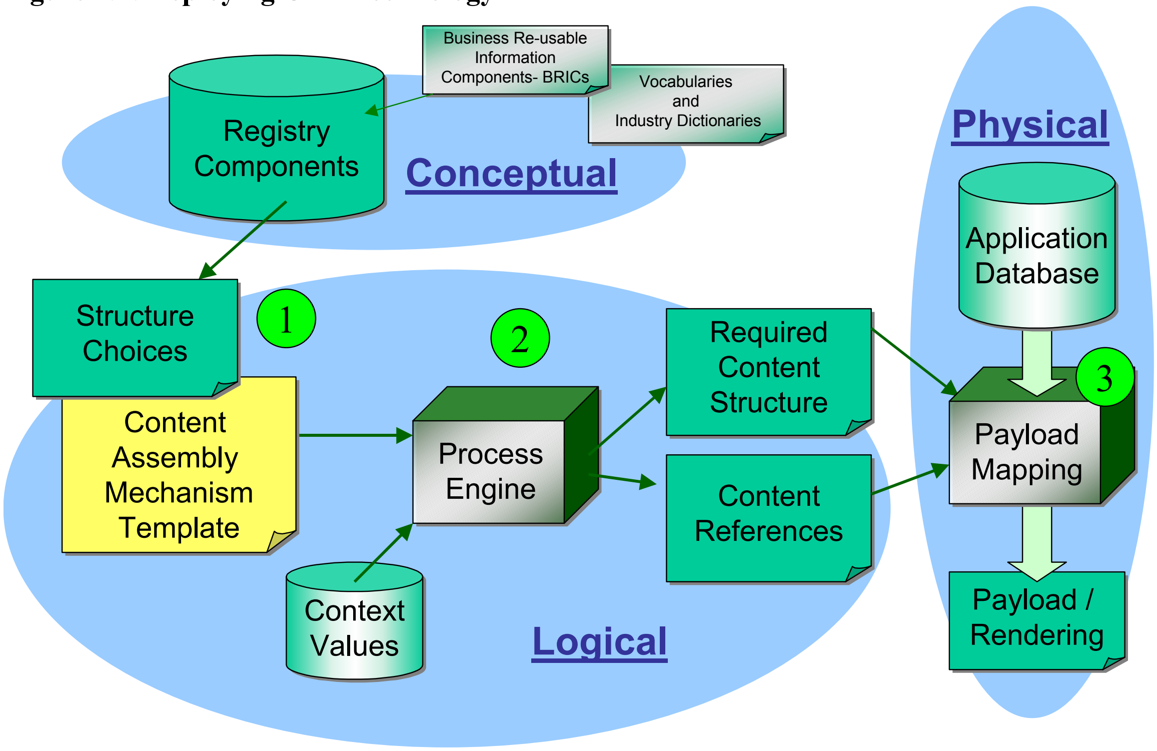
3 Pre-requisites

These specifications make use of W3C technologies, including the XML V1.0, XML namespaces, W3C Schema V1.0 (XSD) with W3C Schema data types V1.0, and XPath 1.0 recommendations. It should be noted that only a subset of the XPath technology, specifically the locator sections of the XPath specification are utilized. Explicit details of XPath syntax are provided in the body of this specification. A schema definition is provided for the assembly mechanism structure. Knowledge of these technologies is required to interpret the XML sections of this document.

4 Content Assembly Mechanism Technical Specification

This section describes the implementation specifications for CAM. Figure 4.1 shows how implementers can integrate CAM technology into their existing systems, and then extend this out to include all aspects of the e-business information content management technologies.

Figure 4.1: Deploying CAM technology



In reference to figure 4.1, item 1 is the subject of this section, describing the syntax and mechanisms. Item 2 is a process engine designed to implement the CAM logic as an executable software component, and similarly item 3 is an application software component that links the e-business software to the physical business application software and produces the resultant transaction payload for the business process itself (these aspects are covered in this document in the addendum on implementation details).

Input to the conceptual model section can come from UML and similar modelling tools to define the core components and relevant re-usable business information components themselves, or can come from existing industry domain dictionaries.

The specification now continues with the detailing the physical realization in XML of the CAM template mechanism itself.

4.1 Overview

The CAM itself consists of five logical sections (as illustrated in figure 2.7.1), and the CAM is expressed in XML syntax. This is shown in figure 4.1.1 as high-level XML structure parent elements[1].

Figure 4.1.1: High-level parent elements of CAM (in simple XML syntax)

```
<CAM CAMlevel="1" version="1.0">
      <Header>
      <AssemblyStructure/>
      <BusinessUseContext/>
      <ContentReference/>
      <DataValidations/>
      <ExternalMapping/>
</CAM>
```

The structure sections provide the ABCDE's of the interchange definition - *A*ssembly Structure(s), *B*usiness Use Context Rules, *C*ontent References (with optional associated data validation), *D*ata Validations and *E*xternal Mappings. Figure 4.1.2[2] next shows the complete hierarchy for CAM at a glance.

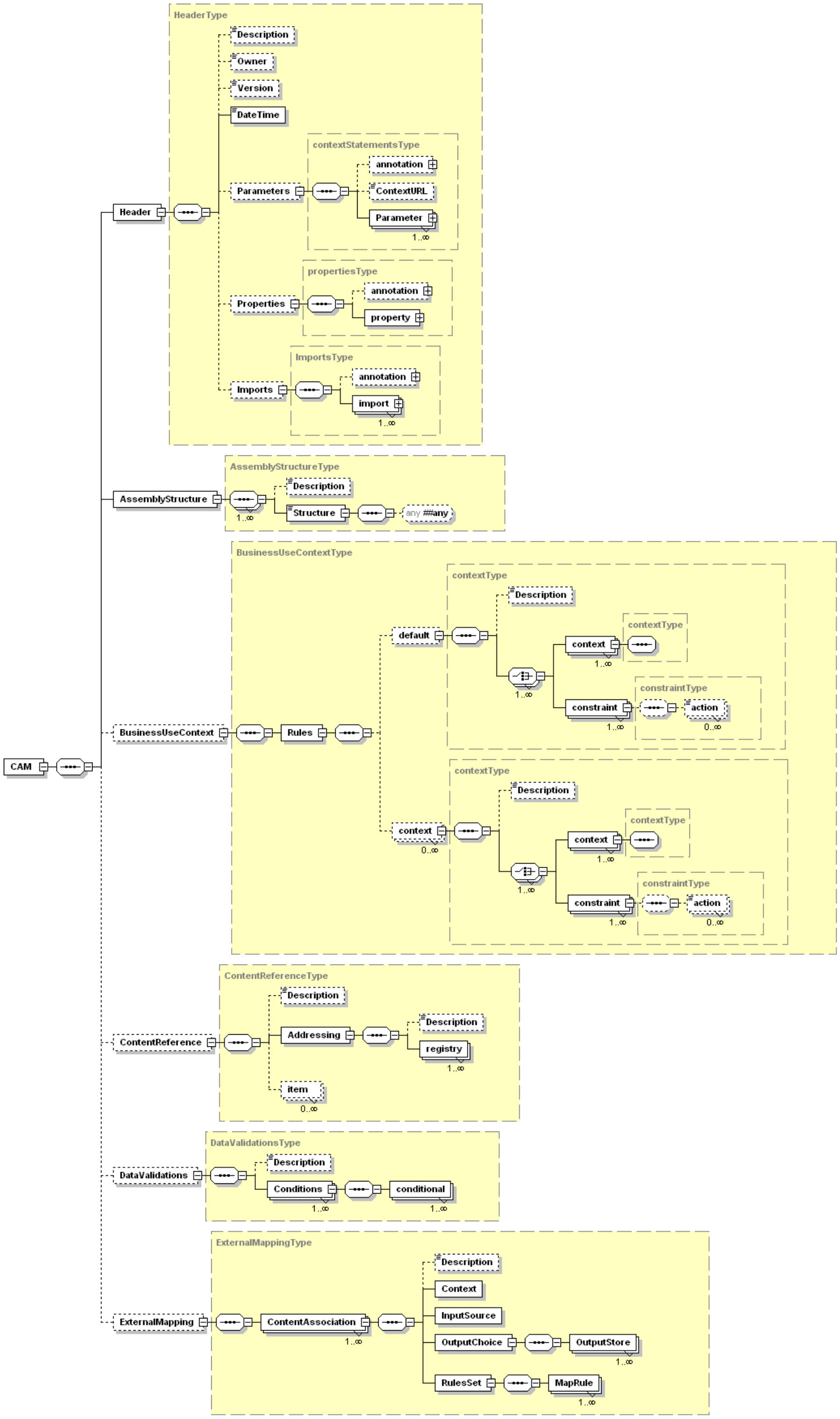
It should be noted that CAM also has built-in compatibility levels within the specification to both aid in implementation of the CAM specification, and also to ensure interoperability.

This is controlled via the CAMlevel attribute of the CAM root element. More details on the CAM implementation levels and features are provided in section 4.8.8 – Conformance Levels and Feature Sets.

[1] Note: elements have been labelled using UN spellings, not North American spellings

[2] This diagrammatic syntax uses modelling notations to show parent, repeated, choice and optional model element linkages. Elements outlined with dashed lines are optional.

Figure 4.1.2: Structure for entire CAM syntax at a glance



Each of these parent items is now described in detail in the following sub-sections, while figure 4.1.3 next shows the formal schema definition for CAM (see the OASIS web site for machine readable Schema formats in XSD syntax). While the documented schema provides a useful structural overview, implementers should always check for the very latest version on-line to ensure conformance and compliance to the latest explicit programmatic details.

The next sections describe each parent element in the CAM in sequence, their role and their implementation details.

4.1.1 Header declarations

The purpose of the Header section is to declare properties and parameters for the CAM process to reference. There are three sub-sections: parameters, properties and imports. Within the main header there are elements that allow documenting of the template description, owner, assigning of a version number and providing a date/time stamp. These are used for informational purposes only and maybe used by external processes to verify and identify that a particular CAM template instance is the one required to be used.

Parameters

This section allows parameters to be declared that can then be used in context specific conditions and tests within the CAM template itself. These can either be substitution values, or can be referencing external parameter values that are required to be passed into this particular CAM template by an external process. External parameters can be passed using the CAM context mechanism (see later section on Advanced Features support). Note: CAM uses the $name syntax to denote external parameter references where required in the CAM template statements.

Properties

These allow creation of shorthand macros that can be referenced from anywhere in the remainder of the CAM template using the ${macroname} reference method. This is designed to provide an easy way to maintain references to external static URL values particularly. It can also be used to define shorthand for commonly repeated blocks of syntax mark-up within the CAM template itself, such as a name and address layout, or a particular XPath expression.

Imports

The import reference allows the CAM processor to pre-load any reference links to external files containing syntax to be included into the CAM template. It also allows the external path of that include file to be maintained in just one place in the template; making easier maintenance if this is re-located. In addition this then allows an <include> statement within the CAM template to reference the import declaration and select a particular sub-tree of content syntax to insert at that given point (using an XPath statement to point to the fragment within the overall import file). This also allows the included content to be done by using just one large file, instead of multiple small files.

The next section begins describing the main processing associated with the CAM template.

4.2 Assembly Structures

The purpose of the AssemblyStructure section is to capture the required content structure or structures that are needed for the particular business process step (i.e. one business process step may have more or more structures it may contextually need to create). This section is designed to be extremely flexible in allowing the definition of such structures. Whereas in this V1.0 specification simple well-formed XML is used throughout to illustrate the usage, for later releases of the CAM specification consideration will be made to allow any fixed structured markup as potentially being utilized as an assembly structure, such as DTD, Schema, EDI, or other (typically they will be used as substitution structures for each other). It is the responsibility of the implementer to ensure that all parties to an e-business transaction interchange can process such content formats where they are applicable to them (of course such parties can simply ignore content structures that they will never be called upon to process).

Notice also that typically a single business process with multiple steps would be expected to have multiple CAM templates, one for each business process step. While it is also possible to provide a single CAM template with multiple structures for a business process with multiple steps, this will likely not work unless the business transaction for each step is essentially the same (since the content reference section and context rules section would have to reference potentially extremely different structures).

Using single CAM templates per step and transaction structure also greatly enhances re-use of CAM templates across business processes that use the same structure content, but different context.

The formal structure rules for AssemblyStructure are expressed by the syntax in figure 4.2.2 below. The figure 4.2.1 here shows a simple example for an AssemblyStructure using a single structure for content.

Figure 4.2.1: Example of Structure and format for AssemblyStructure

```
<Header>
     <Description>Example 4.2.1 using structures</Description>
     <Version>1.0</Version>
</Header>
<AssemblyStructure>
 <Structure taxonomy=”…”>
      <!-- the physical structure of the required content goes here, and can be
      a schema instance, or simply well-formed XML detail, see example below in
      figure 4.2.2 -->
 </Structure >
</AssemblyStructure>
```

In the basic usage, there will be just a single structure defined in the AssemblyStructure / Structure section. However, in the more advanced use, multiple substitution structures may be provided. These can also be included from external sources, with nesting of assemblies; see the section below on Advanced Features for details.

To provide the direct means to express content values within the structure syntax the following two methods apply. A substitution value is indicated by two percentage signs together “%%”, while any other value is assumed to be a fixed content value. Figure 4.2.2 shows examples of this technique.

Figure 4.2.2: Substitution and fixed parameter values, with a well-formed XML structure

```
<Header>
      <Description>Example 4.2.2 Well-formed XML structure</Description>
      <Version>1.0</Version>
</Header>
<AssemblyStructure>
 <Structure taxonomy=”XML”>
   <Items CatalogueRef=”2002”>
       <SoccerGear>
       <Item>
             <RefCode>%%</RefCode>
             <Description>%%</Description>
             <Style>WorldCupSoccer</Style>
             <UnitPrice>%%</UnitPrice>
       </Item>
       <QuantityOrdered>%%</QuantityOrdered>
       <SupplierID>%%</SupplierID>
       <DistributorID>%%</DistributorID>
       <OrderDelivery>Normal</OrderDelivery>
       <DeliveryAddress/>
       </SoccerGear>
   </Items>
 </Structure>
</AssemblyStructure>
```

Referring to figure 4.2.2, the “2002”, “WorldCupSoccer” and “Normal” are fixed values that will always appear in the payload transaction at the end of the CAM process.

In addition to the XML markup, within the AssemblyStructure itself may optionally be included in-line syntax statements. The CAM system provides the BusinessUseContext section primarily to input context rules (see section below), however, these rules may be optionally included as in-line syntax in the AssemblyStructure. However, all rules where present in the BusinessUseContext section take precedence over such in-line syntax rules.

The next section details examples of in-line context rules.

4.3 Business Use Context Rules

Once the assembly structure(s) have been defined, then the next step is to define the context rules that apply to that content. The technique used is to identify a part of the structure by pointing to it using an XPath locator reference, and then also applying an assertion using one of the structure predicates provided for that purpose (an optional comparison evaluation expression can also be used with the XPath locator reference where applicable).

There are two sections to these business context rules, default rules normally apply, and conditional rules that only apply if a particular rule block evaluates to true. The business rules then take the form of structure assertion predicates that define the cardinality of the structure members and content definitions. Figure 4.3.1 shows these structure assertion predicates.

Figure 4.3.1: The assertion predicates for BusinessUseContext

```
excludeAttribute()
excludeElement()
excludeTree()
makeOptional()
makeMandatory()
makeRepeatable()
setChoice()
setId()
setLength()
setLimit()
setRequired()
setMask()
setValue()setUID()
restrictValues()
restrictValuesByUID()
useAttribute()
useChoice()
useElement()
useTree()
useAttributeByID()
useChoiceByID()
useElementByID()
useTreeByID()
startBlock()
endBlock()
checkCondition()
makeRecursive()
```

Each predicate provides the ability to control the cardinality of elements within the structure, or whole pieces of the structure hierarchy (children within parent).   An example of such context rules use is provided below, and also each predicate and its' behaviour is described in the matrix in figure 4.3.3 below. Also predicates can be used in combination to provide a resultant behaviour together, an example is using makeRepeatable() and makeOptional() together on a structure member.

Note that the BusinessUseContext section controls use of the structure, while if it is required to enforce explicit validation of content, then there is also the DataValidations section that provides the means to check explicitly an element to enforce content rules as required.   See below for details on this section. This validation section is also further described in the advanced use section since it can contain extended features.

Predicates that affect the definition of the content that will be used in any context is derived by applying the rules using the following precedence rules. The lower numbered rules are applied first and can be overridden by the high numbered rules.

1. AssemblyStructure Inline predicates.
2. ContentReference predicates.
3. BusinessUseRules default rules and predicates.
4. BusinessUseRules conditional rules and predicates.

Referring to the structure in the example shown in figure 4.2.2, figure 4.3.2 provides examples of context based structural predicate assertions. Notice that such context rules can be default ones that apply to all context uses of the structure, while other context rules can be grouped and constrained by a XPath locator rule expression. There are three styles of such XPath expressions:

1. XPath expression refers to structure members directly and controls their use
2. XPath expression refers to structure member and contains condition of its value
3. XPath expression refers to token that is not member of structure, but is a known external control value from the profile of the business process itself.

Such XPath expressions will match all the structural elements that they can refer to, so if a unique element is always required, implementers must ensure to provide the full XPath identity so that only a single unique match occurs. An example is a reference to "//ZIPCode" which will match any occurrence, whereas "/BillingAddress/ZIPCode" will only match that item.

Figure 4.3.2: Syntax example for BusinessUseContext

```
<BusinessUseContext>
<Rules>
   <default>
     <context> <!-- default structure constraints -->
      <constraint action="makeRepeatable(//SoccerGear)" />
       <!—type 1 Xpath-->
      <constraint action="makeMandatory(//SoccerGear/Items/*)" />
```

```
       <constraint action="makeOptional(//Description)" />
       <constraint action="makeMandatory(//Items@CatalogueRef)" />
       <constraint action="makeOptional(//DistributorID)" />
       <constraint action="makeOptional(//SoccerGear/DeliveryAddress)" />
     </context>
   </default>
   <context condition="//SoccerGear/SupplierID = 'SuperMaxSoccer'">
       <!—type 2 Xpath-->
     <constraint action="makeMandatory(//SoccerGear/DeliveryAddress)"/>
   </context>
   <context condition="$DeliveryCountry = 'USA'">
       <!—type 3 Xpath-->
     <constraint action="useTree(//SoccerGear/DeliveryAddress/USA)"/>
   </context>
</Rules>
</BusinessUseContext>
```

Referring to the XPath expressions in figure 4.3.2, examples of all three types of expression are given to show how the XPath expressions are determined and used. For external control values the special leading $ indicator followed by the variable name denotes a substitution value from a context reference variable that is declared in the CAM template header.

Referring to figure 4.3.3 below, the following applies:

//elementpath XPath expression resolving to an element(s) in the structure. This parameter is not required when predicate is used in-line, since then it is implicit.

//memberpath XPath expression resolving to either an element(s) or an attribute(s) in the structure

//treepath XPath expression resolving to parent element with children in the structure

//StructureID reference to an in-line ID assignment within the structure, or ID value assigned using setID() predicate.

//elementpath@attributename — XPath expression resolving to an attribute or attributes in the structure

//attributepath — This can be used interchangeably with //elementpath when //memberpath is an allowed parameter of a predicate. Either a single XPath expression resolving to an attribute in the structure, or a collection of XPath expressions referencing more than one attribute for the given element of the form //elementpath@[attributename1, attributename2, attributename3,…], or //elementpath@[*] to reference all attributes for that element.

IDvalue — String name used to identify structure member

UIDreference — Valid UID and optional associated registry and taxonomy that points to an entry in a Registry that provides contextual metadata content such as a [valuelist] or other information

value, valuelist, count, mask — String representing parameter. When lists are required then group with paired brackets [a, b, c, …], and when group of groups use nested brackets [[a, b, d, f],[d, e, g, m]]

Note: groups are required for collections of attributes in in-line predicate assertions.

Figure 4.3.3: Matrix of predicates for BusinessUseContext declarations.

Predicate	Parameter(s)	Description
excludeAttribute()	//elementpath@attributename	Conditionally exclude attribute from structure
excludeElement()	//elementpath	Conditionally exclude element from structure
excludeTree()	treepath	Conditionally exclude a whole tree from structure
makeOptional()	//elementpath	Conditionally allow part of structure to be optional
makeMandatory()	//elementpath	Conditionally make part of structure required
makeRepeatable()	//elementpath	Conditionally make part of structure occur one or more times in the content
setChoice()	//elementpath	Indicate that the first level child elements below the named elementpath are actually choices that are conditionally decided with a useChoice() predicate action
setId()	//elementpath,IDvalue	Associate an ID value with a part of the structure so that it can be referred to directly by ID
setLength()	//memberpath, value	Control the length of content in a structure member
setLength()	//memberpath, [value-value]	Control the length of content in a structure member, allows two factors for range of lengths.
setLimit()	//elementpath, count	For members that are repeatable, set a count limit to the number of times they are repeatable

Predicate	Parameter(s)	Description

`setMask()`	`//memberpath, datatype, [mask | masklist]`
	or
	`//memberpath, [mask | masklist]`
		Assign a CAM picture mask to describe the content. The mask can also set explicit datatype of an item as well using the first parameter of the mask accordingly (default is string if datatype parameter omitted). Masklist allows an optional list of masks to be provided as well as one single mask.

`datatype()`
or
`setDatetype()`
	`//memberpath, value`	associate datatype with item, valid datatypes are same as W3C datatypes. If a setMask() statement is present for the item, this statement will be ignored.

`setRequired()`	`//elementpath,value`	For members that are repeatable, set a required occurrence for the number of members that must at least be present (nnnn must be greater than 1)[3].
`setValue()`	`//memberpath, value`	Place a value into the content of a structure
`setValue()`	`//memberpath, [valuelist]`	Place a set of values into the content of a structure (allows selection of multiple values of member items).
`setUID()`	`//memberpath, alias, value`	Assign a UID value to a structure element. Alias must be declared in registry addressing section of ContentReferences).
`restrictValues()` or	`//memberpath, [valuelist],[defaultValue]`	Provide a list of allowed values for a member item

[3] Design note: makeRepeatable(), makeMandatory() is the preferred syntax over the alternate: makeRepeatable() as:setRequired="1".

Predicate	Parameter(s)	Description
`member()`		

`restrictValuesByUID()`
or
`memberByUID()`
	`//memberpath, UIDreference,`
	`[defaultValue]`
		Provide a list of allowed values for a member item from a registry reference

`useAttribute()`	`//elementpath@attributename,`
	`or //attributepath`
		Require use of an attribute for a structure element and exclude other attributes

`useChoice()`	`//elementpath`	Indicate child element to select from choices indicated using a setChoice() predicate.
`useElement()`	`//elementpath`	Where a structure definition includes choices indicate which choice to use (this function is specific to an element path, and does not require a prior setChoice() predicate to be specified).
`useTree()`	`//treepath`	Where a structure member tree is optional indicate that it is to be used. Note: the //treepath points directly to the parent node of the branch and implicitly the child nodes below that, that are then selected.
`useAttributeByID()`	`StructureID`	As per useAttribute but referenced by structure ID defined by SetId or in-line ID assignment
`useChoiceByID()`	`StructureID`	As per useChoice but referenced by structure ID defined by SetId or in-line ID assignment
`useTreeByID()`	`StructureID`	As per useTree but referenced by structure ID defined by SetId or in-line ID assignment
`useElementByID()`	`StructureID`	As per useElement but

Predicate	Parameter(s)	Description
		referenced by structure ID defined by SetId or in-line ID assignment
checkCondition()	conditionID	conditionID is required and references the ID of the conditional block in the data validation section (defined in attribute – conditioned). The validation block will be performed at that point in the structure processing flow.
makeRecursive()	StructureID	Denote that the specified parent element can occur recursively as a child of this parent.
startBlock() Advanced Option	StartBlock, [StructureID]	Denote the beginning of a logical block of structure content. The StructureID is an optional reference. This function is provided for completeness. It should not be required for XML structures, but may be required for non-XML content; basic CAM conformance at Level 1 does not require this function.

Predicate	Parameter(s)	Description

`endBlock()`
`Advanced Option`
	`endBlock, [StructureID]`	Denote the end of a logical block of structure content. The StructureID is an optional reference, but if provided must match a previous startBlock() reference. This function is provided for completeness. It should not be required for XML structures, but may be required for non-XML content; basic CAM conformance at Level 1 does not require this function.

`lookup ()`
`Advanced Option`
	`lookup (valuelist, ’call address’)`	Conditionally check for a string being located in a list referenced by a call address. *Note: call address is defined in ContentReference section.* More than one value may be passed for associated codelists.

`memberReplace ()`
`Advanced Option`
	`member (valuelist,`
	`‘[value,value,value,…]’,`
	`‘[replace,replace,replace,…]’)`
		As with member(), but returns a matching replacement value from the same position in the third parameter.

The predicates shown in figure 4.3.3 can also be used as in-line statements within an assembly structure, refer to the section on advanced usage to see examples of such use.

4.3.1 XPath syntax functions

The W3C XPath specification provides for extended functions. The CAM XPath usage exploits this by following the same conditional evaluations as used in the open source project for the jaxen parser (this is used as the reference XPath implementation). The base XPath provides the “contains” function for examining content, the jaxen functions shown in figure 4.3.4 extend this to provide the complete set of familiar logical comparisons.

Figure 4.3.1.1 XPath Comparator functions.

Comparator	Syntax	Description
Equal to	$variable = 'testValue'	Conditionally check for a matching value
Not equal to	not(value1,'value')	Conditionally check for a non-matching value
Greater than	value > value or value &gt; value	Conditionally check for a greater value
Less than	value < value or value &lt; value	Conditionally check for a lesser value
Greater than or equal	value >= value or value &gt;= value	Conditionally check for a greater than or equal to value
Less than or equal	Value <=value or value &lt;= value	Conditionally check for a lesser or equal value
begins	starts-with(value,value)	Conditionally check for a string matching the front part of value, equal or longer strings match.
ends	ends-with(value,value)	Conditionally check for a string matching the end part of value, equal or longer strings match.
String length	string-length()	Conditional check for the length of a string.
Count	count()	Conditionally check for the occurrence of an element
Contains	contains (value,‘value’)	Conditional check for an occurance of one string within another.
concat	Concat(//elementpath, //elementpath, ‘stringvalue’)	The ‘+’ operator concatenates the values from locators together as a string, or constant string values. This allows evaluations where the content source may separate related fields; e.g. Month, Day, Year.

Using these capabilities provides sufficient expressive capability to denote structural combinations for context driven assembly and also for basic data validation (see following applicable sections).

The next section shows how to associate a reference to a dictionary of content model metadata, or to provide the content model directly for members of the structure content.

4.3.2 Handling CDATA content with XPath

An XML element parent may enclose a CDATA section of embedded information. When outputting such information there are two choices, the CDATA markup may be stripped off and the data processed, or the CDATA section, including the markup, is passed through "as-is" into the output. The XPath expression can only reference the parent element and not any markup within the CDATA itself.

4.4 CAM character mask syntax

In order to provide a base-line character mask set, and also to provide a character mask set that is accessible to business technical users as well as programming staff, the following is provided as a default character mask system. This mask system is based on that used by typical program generator tools available today and is designed to provide a neutral method that can be mapped to specific program language syntax as needed. The mask system syntax is provided in Addendum section A.1.6 and usage details are also provided there and can be found by studying the examples provided in the example tables. (Note: consideration of alternate mask systems being specified in other syntaxes such as SQL, Perl, and so on will be added for later versions of CAM).

Description

Picture masks are categorized by the basic data-typing element that they can be used in combination with. Content that already conforms to the mask is not modified but simply placed in the DOM as is. Content that does not conform to the mask (such as text in a numeric field) results in '*' characters being placed in the DOM to the full length of the mask, so 'ABC' in a field defined as #6.## would result in '*********', and so on.

The first parameter of the mask indicates the types. Valid values are any W3C data type such as: string, decimal, integer, datetime, time, date, binary and three additional CAM data types of email (a valid email address format), logical (Boolean), and filepath (a valid operating system file path).

Note for items of arbitrary length and no mask – use the datatype() function instead of setmask().

String Pictures

The positional directives and mask characters for string pictures are as follows:
X - any character mandatory
? – any character optional, * - more than one character, arbitrary occurrence of – (equivalent to CDATA).
U - a character to be converted to upper case
^ - uppercase optional
L - a character to be converted to lower case
_ - Lowercase optional

0 - a digit (0-9 only)
# - a digit (0-9 only), trailing and leading zeros shown as absent
Examples of string pictures are shown in the following table:

String value	Picture mask (shorthand)	Full expanded mask	Resulting string value
portability	X6	XXXXXX	portab
portability	UX3	UXXX	Port

portability	XXXXing	XXXXing	porting
realtime	XXXX-XXXX	XXXX-XXXX	real-time
BOLD!	L5	LLLLL	bold!

Numeric Pictures

The positional directives and mask characters for numeric pictures are as follows:
0 - a digit (0-9 only)
# - a digit (0-9 only), trailing and leading zeros shown as absent
. – indicates the location of the decimal point. For example, '0000.000' defines a numeric variable of four whole digits and three decimal digits

Examples of numeric pictures are shown in the following table (the ^ symbol represents one space character):

Numeric value	Picture	Resulting numeric value
-1234.56	######.##	-1234.56
-1234.56	000000.##	-001234.56
-1234.56	N######.##	^^-1234.56
-1234.56	N###,###.##C	^^-1,234.56
-1234.56	N######.##L	-1234.56^^
-1234.56	N######.##P*	-**1234.56
0	N######.##Z*	*********
-13.5	N##.##-DB;	DB13.50
45.3	N##.##+CR;	CR45.30
-13.5	N##.##-(,);	(13.50)
4055.3	$######.##	$^^4055.30

Date Pictures

The typical date formats are DD/MM/YYYY (European), MM/DD/YYYY (American), or YYYY/MM/DD (Scandinavian). When you define the attribute Date for a variable, you must also select the format for the date item (see below). You can change this default picture and place in it any positional directives and mask characters you need.

DD—A place holder for the number of the day in a month
DDD—The number of the day in a year
DDDD—The relative day number in a month
MM—A place holder for the number of the month in a year
MMM...—Month displayed in full name form (up to 10 'M's in a sequence). e.g. January, February. If the month name is shorter than the number 'M's in the string, the rest of the 'M' positions are filled with blanks.
YY—A place holder of the number of the year
YYYY—A place holder for the number of the year, represented in full format (e.g. 1993)
W—Day number in a week
WWW...—Name of day in a week. The string can be from 3 to 10 'W's. If the name of the day is shorter than the number of 'W's in the string, the rest is filled with blanks.
/—Date separator position.
-—Date separator position (alternate).

Examples of date pictures are shown in the following table, using the date of 21 March 1992 (the ^ symbol represents one space character – used to show spaces for this document only):

Picture	Result and notes
MM/DD/YYYY	03/21/1992
##/##/##	Note: 21/03/92, when XML parser default is set to European, 03/21/92, when XML parser is set to American
MMMMMMMMMM^DDDD, ^YYYY	March^^^^^^21st,^1992
MMMMMMMMMM^DDDD, ^YYYYT	March^21st,^1992 with trimming directive (see below)
WWWWWWWWWWW^-^W	Saturday^^^-^7
WWWWWWWWWWW^-^WT	Saturday^-^7 with trimming directive (see below)

"Trimming directive" is invoked by adding the directive T to the variable picture. This directive instructs XML parser to remove any blanks created by the positional directives 'WWW...' (weekday name), 'MMM...' (month name), or 'DDDD' (ordinal day, e.g. 4th, 23rd). Since these positional directives must be specified in the picture string using the maximum length possible, unwanted blanks may be inadvertently created for names shorter than the specified length. The Trim Text directive will remove all such blanks. If a space is required nevertheless, it must be explicitly inserted in the picture string as a mask character, (the ^ symbol is used to indicate a blank character), e.g., 'TWWWWWWWWWW^DDDD MMMMMMMMM,^YYYY'

"Zero fill" is invoked by adding the functional directive Z to the variable picture. This directive instructs XML parser to fill the entire displayed variable, if its value is zero, with the "Character" value. If you don't specify a Character the variable is filled with blanks.

Time Pictures

The XML parser defines the default picture mask HH/MM/SS for an element of datatype Time. Examples of time pictures are shown in the following table:

Picture	Result	Comments
HH:MM:SS	08:20:00	Time displayed on 24-hour clock.
HH:MM:SS	16:40:00	Time displayed on 24-hour clock.
HH:MM PM	8:20 am	Time displayed on 12-hour clock.
HH:MM PM	4:40 pm	Time displayed on 12-hour clock.
HH-MM-SS	16-40-00	Using Time Separator of '-'

4.5 Content Referencing

The purpose of content referencing is to provide additional information about the metadata of each item of the structure, the content model, and associated data typing when applicable. It also provides crosswalk information to a dictionary of noun definitions, and thus potentially from your physical implementation to the logical aggregate component themselves. This ability to provide crosswalk implementation details is vital to maximizing interoperability and re-use within the optimal e-business architecture and also allowing the use of modelling tools and object-oriented technologies.

The example in figure 4.4.1 shows the content referencing for the structure in figure 4.2.2, and shows how multiple dictionary domains (namespaces) can be accommodated in blending a composite structure together, while also allowing extensions using locally defined content items that are not part of any dictionary. The use cases for content referencing can be summarized as:

1. No registry dictionary is available so all content referencing is locally defined
2. A default content model can be defined using the predicates, (these however will not take precedence over explicit rules in the BusinessContext section), but will override any inline predicates within AssemblyStructure
3. A single registry and industry domain is referenced only
4. Multiple registry domains are referenced
5. Combinations of all of the above

Further notes on aspects of the particular syntax instructions for content referencing are given below.

Figure 4.4.1: Example of Content Referencing for AssemblyStructure

```
<ContentReference>
   <Addressing>
       <registry name="SGIR" access="registry.sgir.org:1023" method="URL"
                     description="Sporting Goods Industry Registry"/>
       <registry name="SGIRWSDL" access="registry.sgir.org:1025" method="WSDL"
                     description="Sporting Goods Industry Registry"/>
       <registry name="UN" access="registry.un.org:9090" method="ebXML"
                     description="United Nations EDIFACT Registry"/>
       <registry name="UPS" access="registry.ups.com:7001" method="URL"
                     description="United Parcels Service Registry"/>
       <registry name="USPS" access="registry.usps.gov:8080" method="URL"
                     description="United States Postal Service Registry"/>
       <registry name="LocalSQL" access="rdbms.mybusiness.com:4040" method="SQL"
                     description="Local Product Database stored procedures"/>

   </Addressing>

   <item type="noun" name="RefCode"
             UIDReference="SGIR010027" taxonomy="UID" registry="SGIR"/>
   <item type="noun" name="Description"
             UIDReference="SGIR010050" taxonomy="UID" registry="SGIR"/>
   <item type="noun" name="Style"
             UIDReference="SGIR010028" taxonomy="UID" registry="SGIR"/>
   <item type="noun" name="SupplierID"
             UIDReference="SGIR010029" taxonomy="UID" registry="SGIR"/>
   <item type="noun" name="CatalogueRef" UIDReference="none" taxonomy="none"
             datatype="string" setlength="4" setMask="p\d\d\d\d" />
   <item type="noun" name="DistributorID" UIDReference="none" taxonomy="none"
             datatype="string" setlength="30" />
   <item type="noun" name="UnitPrice"
             UIDReference="070010" taxonomy="EDIFACT" registry="UN"/>
   <item type="noun" name="QuantityOrdered"
             UIDReference="070011" taxonomy="EDIFACT" registry="UN"/>
   <item type="noun" name="OrderDelivery"
             UIDReference="UPS050050" taxonomy="UID" registry="UPS"/>
   <item type="defaultAssembly" name="DeliveryAddress"
             UIDReference="USPS090081:01:05" taxonomy="UID" registry="USPS"/>
 </ContentReference>
```

Each of the modes of determining a content reference is shown in figure 4.4.1, along with the use of the Registry addressing section to link between the logical and physical addresses of Registry content. Notice that with locally defined items (UIDReference="none" taxonomy="none"), then one of the optional predicate[4] parameters is used to further define the content model (e.g.: setlength="4").

Typically references are to nouns within the assembly structure, but can also be to a composite item as a defaultAssembly, as is the case with the DeliveryAddress example (such defaultAssembly items can equate to aggregate components, and have an <as:include> for their structure content, see details in the advanced techniques section below).

Similarly the taxonomy preferred is that of the UID system, however where legacy schemes exist such as EDI element dictionary numbering, then the UIDReference can accommodate such values accordingly. The UID values themselves are composed of an alpha prefix representing an acronym for the domain organization, followed by a simple 6-digit numeric. Optionally a UID can also have a suffix of colon, major version, and colon, minor version, to provide version control. When the version information is omitted then the UID reference points to the latest current information from the registry by default.

If an item refers to a registry acronym that is not defined in the //Addressing/registry statement, then a warning should be issued, but processing can continue. Similarly, warnings should be generated for assembly structure members that do not have ContentReference entries, but all such items will have a default content model of type="text" as a simple string type. Notice that type="[datatype]" supports the W3C Schema data types by default.

The content referencing is intended to provide assembly metadata for the information content model during assembly. The next section can handle post-assembly processing and validation requirements on receipt of content, as well as on creation of content.

4.6 Data Validations

This section provides the means to verify information content of transaction instances built from CAM structure and context rules. This is an advanced option. This verification can occur at design/runtime during creation of a content instance, and also some verification can occur after post-content creation, typically upon content receipt by some other party. The DataValidation section is thus more likely to be tied to a particular production implementation and environment, particularly for post-content creation checks. However, users can choose to provide generic CAM formulas that apply to all implementations within a domain using XPath expressions as allowed within CAM, and then allow implementers to extend these for particular local instances.

Validation rules are allowed only using CAM compatible XPath expressions or calls via the Registry call mechanism defined within the Content Reference section.

[4] Implementation note: the XPath parameter for the predicate defaults to the name value to identify the item within the assembly structure

Execution of data validations occurs after processing of all the preceding sections in the CAM template. However, processing of data validation conditions may occur during structure processing if an explicit checkCondition() instruction is used inline (see advanced techniques section below) and that references a condition block by conditionID. Any checks in the data validation section itself that are labelled with a conditionID will be skipped when processing proceeds to the DataValidation section itself. This allows data validations to be invoked where needed; either inline within a structure, or from the business context rules section, or at the end of processing of an XML record block (the normal sequence).

Figure 4.5.1: Example of Data Validations for AssemblyStructure

```
<DataValidations>
   <Conditions
       conditionID="testOrderDetail"
       condition="$DeliveryCountry = 'USA'">
       <conditional
             expression="’//UnitPrice’ and greaterthan(value,’0.00’)"
             syntax="XPath" outcome="fail"
             message="Item price not valid / missing" test="always"/>
       <conditional
             expression="’//RefCode + //UnitPrice’ and
             lookup(value,’SGIRWSDL:unitprice_check’)” outcome="report"
             message="Unit price value does not match catalog" test="always"/>
       <conditional
             expression="’//SupplierID’ and
             lookup(value,’SGIRWSDL:supplierID_check’)” outcome="fail"
             message="Unknown Supplier ID" test="always"/>
       <conditional
             expression="’//DistributorID’ and
             lookup(value,’SGIRWSDL:distributor_check’)” outcome="fail"
             message="Unknown distributor ID" test="postcheck"/>
       <conditional itemRef="//QuantityOrdered"
             expression="’//QuantityOrdered’ and
             lookup(value,’LocalSQL:quantityOnHand()’)” outcome="report"
             message="Item not available / backordered" test="postcheck"/>
   </Conditions>
</DataValidations>
```

The conditional section shown in figure 4.5.1 shows a variety of methods, from in-line XPath expressions, remotely executed ‘verbs’ from a registry as a web service, to SQL stored procedures. Notice that WSDL is used as the interface example to web services, and the WSDL description may involve passing of parameters (such as the //RefCode to verify the //UnitPrice).

These details can be determined through the programmatic interface to the particular lookup reference service[5].

Again, support for these methods is dependent on the business agreements between parties and the capabilities and requirements of parties. Some parties may simply opt to not support DataValidation conditions, or only those using XPath, and so on. Because of this, it is anticipated that the DataValidation section will provide useful hints to parties on requirements for a complete and accurate business exchange. How far they will be able to support these, and how many local extensions are built using the base mechanisms provided in the syntax methods of DataValidation will depend on the maturity of the information systems of the implementers. Since these mechanisms and section are least accessible to business users, and most accessible to programmers the initial intent here is to provide basic functionality that is useful to a broad range of business use. It is not intended to replace extensive, proprietary and complex application logic in backend systems.

For a simple implementation it is suggested that basic information checks are instituted using the provided XPath syntax and comparator functions. Then later more extended checks can be supported via external calls. Similarly if the outcome is marked as ‘ignore’ or ‘report’, this means that early implementers can treat these checks simply as documentation notes as to the checking that backend complex application logic will perform, until they are more fully able to support the recovery and post-processing required via their business processing service components.

4.6.1 Discrete Value List Support (“Codelists”)

This note discusses support for code list functionality. Over 50% of traditional EDI transaction content is comprised of code values that are referenced and shared between trading partners. CAM provides two XPath functions to directly implement these capabilities. Firstly is the `member()` function that allows specific code values to be specified in the CAM template itself. Second is the `lookup()` function that supports the use of code values external to the template itself, where one or more parameters are passed into it. Configuration of the lookup function external access is achieved through the Content Reference section Registry definition statements. See the examples provided in that section, and in the validation examples in figure 4.5.1 above. Nested code list lookups can be configured using nested `<conditions>` expressions.

Also versioning of codelist lookups can be achieved through the version mechanism on the UID reference mechanism when using codelists retrieved from a registry system. When codelist values are provided as in-line static lists, then selection can be achieved by providing choices of structure items driven off a context variable and the use of `choiceID()` predicates.

Similarly if context driven selection of codelist values is required it can also be implemented with `choiceID()` predicates selecting `lookup()` functions with static lists of values.
The next section details further advanced features that can be used to augment the basic CAM functionality.

[5] Note: OASIS Registry support for CAM services through this specification is covered separately in the addendum of this specification document.

4.7 External Business Content Mapping

The business content mapping is an optional component to the base assembly functionality, and is primarily intended to bridge between the neutral assembly approach and specific domain implementations. The business content mapping script instructions are designed to provide non-procedural hints to implementation systems. Implementers can choose to use these to drive specific back-end application systems, or simply as documentation to constructing such application system linkages within their own systems. This can then provide useful hints to the assembly process itself or to implementations integrating multiple application systems and requiring extended crosswalk information. Included in this is the ability to merge content into a static target structure by using a set of merge commands for token replacements. In this instance the external mapping rules bridge between the input source data and the target merge structure replacement token names (or “mail-merge” style replacement).

This initial release is a simple non-procedural system that allows specification of statements that can bridge between the assembly transaction and the business application. It is not intended to provide a complete full-function computation engine, but does provide the ability to simply equate between application content and structure content members with some ability to manipulate the content (it should be noted to that XPath statements contain some limited content manipulation functionality as well).

There are two styles that external content mapping therefore supports. The first is illustrated by example 4.10.1 where the content output is in formal location (table) / columnar / row formatting typical of database SQL processing. The other approach is for semi-structured output based on tokenised fields into some target structure format, and multiple such fixed formats may be specified based on a context variable choice as required. This second approach is designed to accommodate outputting into formats such as xhtml, XForm, or a transaction structure such as XML or EDI targets.

Figure 4.6.1: Example of business content mapping script to a columnar output format

```
<ExternalMapping>
 <ContentAssociation>
  <Description>Product List</Description>
  <Context/>
  <InputSource/>
  <OutputStore type=”SQL” location=”product_table”/>
  <RulesSet>
   <MapRule output="Products_List" input="@STARTGRP()"/>
       <MapRule output="type" input="Sales/Company/Year/Qtr/Product@type"/>
       <MapRule output="name"
             input="@trim(Sales/Company/Year/Qtr/Product/Item@name)"/>
       <MapRule output="manufacturer"
```

```
             input="Sales/Company/Year/Qtr/Product/Item@manufacturer"/>
       <MapRule output="value"
             input="Sales/Company/Year/Qtr/Product/Item@value"/>
       <MapRule output="sold"
             input="Sales/Company/Year/Qtr/Product/Item@sold"/>
   <MapRule output="Products_List" input="@ENDGRP()"/>
  </RulesSet>
 </ContentAssociation>
</ExternalMapping>
```

The syntax for this section is summarized in the table shown in figure 4.10.2. These predicates are designed as a simple set of sparse commands that augment the XPath statements to provide a core of content string based functionality.

Figure 4.6.2: Summary of business content mapping script commands

Predicate	Parameter(s)	Description
`@concat(p1,p2)`	`[//memberpath | string | predicate()], [//memberpath | string | predicate()]`	Combine two strings together. Predicates can be combined to derive resultant string content.
`@trim(p1)`	`//memberpath`	Remove trailing and leading white space from content.
`@startgrp()`	`[//memberpath]`	Start of a loop of recurring content. Optional memberpath reference denotes when 'next record' condition occurs on change of value / occurance in the input structure.
`@endgrp()`	`None`	End of a loop of recurring content
`@multiply(p1,p2)`	`[//memberpath | string | predicate()], [//memberpath | string | predicate()]`	Compute result of calculation; see arithmetic note at end of table.
`@divide(p1,p2)`	`[//memberpath | string | predicate()], [//memberpath | string | predicate()]`	Compute result of calculation; see arithmetic note at end of table.

Predicate	Parameter(s)	Description
`@add(p1,p2)`	`[//memberpath | string | predicate()], [//memberpath | string | predicate()]`	Compute result of calculation; see arithmetic note at end of table.
`@subtract(p1,p2)`	`[//memberpath | string | predicate()], [//memberpath | string | predicate()]`	Compute result of calculation; see arithmetic note at end of table.
`@if(p1,p2,p3)`	`Expression, [//memberpath | predicate()], [//memberpath | predicate()]`	Logical expression, if the conditional expression is true, then p2, else p3.
`@upper(p1)`	`[//memberpath | string | predicate()]`	Change all characters to their uppercase equivalent (works only for a string of Latin, Cyrillic or Greek characters: for most other languages case is irrelevant. See ISO 10646 as reference behaviour here).
`@lower(p1)`	`[//memberpath | string | predicate()]`	Change all characters to their lowercase equivalent, (works only for a string of Latin, Cyrillic or Greek characters: for most other languages case is irrelevant. See ISO 10646 as reference behaviour here).
`@len(p1)`	`[//memberpath | string | predicate()]`	Returns length of string item.
`@left(p1,p2)`	`[//memberpath | string | predicate()],[numeric | //memberpath | predicate()]`	Return p2 number of leftmost characters from a string p1.
`@right(p1,p2)`	`[//memberpath | string | predicate()],[numeric | //memberpath | predicate()]`	Return p2 number of rightmost characters from a string p1.
`@mid(p1,p2,p3)`	`[//memberpath | string | predicate()],[numeric | //memberpath | predicate()],[numeric | //memberpath | predicate()]`	Return p3 number of characters from a string p1 starting from position p2.

835

Note on parameters to arithmetic functions: p1 and p2 must be valid datatypes of either integer, or decimal. If one factor is decimal, that precision will be used for the result. If one or both of the parameters are not valid numeric values, then the function will cause any conditional expression to evaluate to 'false'.

4.8 Advanced Features

This section details extended uses of the basic features. For this first release this is focused on three aspects, in-line use of predicates within structures, non-XML structure content referencing, and external content inclusion into a CAM. To help with configuring and controlling advanced features the properties section has now been added to the CAM structure. This allows programmatic control syntax to be added easily in the future to support advanced feature configuration options.

4.8.1 In-line use of predicates and references

Figure 4.8.1.1 shows an extended example for an AssemblyStructure using two different structures for content and the in-line statements indicating those content selections. The in-line commands are inserted using the "as:" namespace prefix, to allow insertion of the command statements wherever they are required.   These in-line commands compliment the predicates used within the <BusinessUseContext> section of the assembly. The table in figure 4.7.1.2 gives the list of these in-line statements and the equivalent predicate form where applicable.

Figure 4.7.1.1: Example of Multiple substitution structures for AssemblyStructure

```
<CAM CAMlevel="1" version="1.0"
 xmlns:as="http://www.oasis-open.org/committees/cam">
 <AssemblyStructure>

   <Structure as:choiceID="FirstOne" taxonomy='XML'>
         <!-- the physical structure of the required content goes here -->
   </Structure >

   <Structure as:choiceID="SecondOne" taxonomy='XML'>

        <createTroubleTicketByValueResponse as:choiceID="OptionA">
             <!-- the physical structure of the required content goes here -->
        </createTroubleTicketByValueResponse>

        <createTroubleTicketByValueResponse as:choiceID="OptionB">
             <!-- the physical structure of the required content goes here -->
        </createTroubleTicketByValueResponse>
   </Structure >

 </AssemblyStructure>
</CAM>
```

Reviewing figure 4.7.1.1 there are two main substitution structures, and within the second there are also two sub-structure choices. The actual behaviour and which structure content is included in the physical content is controlled by predicate statements within the <BusinessUseContext> section of the assembly.

The in-line statements available are detailed in the table shown in figure 4.5.1.2. In-line command entries marked as "not applicable" can only be used within the <BusinessUseContext> section. Also where there is both a predicate statement and an in-line command, then the predicate statement overrides and takes precedent.

Figure 4.7.1.2: Matrix of in-line statement commands and predicate commands.

Predicate	In-line Command	Notes
`excludeAttribute()`	`Not applicable`	
`excludeElement()`	`Not applicable`	
`excludeTree()`	`Not applicable`	
`makeOptional()`	`as:makeOptional="true"`	Make part of structure optional, or make a repeatable part of the structure optional (e.g. occurs=zero)
`makeMandatory()`	`as:makeMandatory="true"`	Make part of the structure required

`makeRepeatable()`	`as:makeRepeatable="true"`
	`as:setLimit="5"`
	`as:setRequired="3"`
		Make part of the structure occur one or more times in the content; the optional as:setLimit="nnnn" statement controls the maximum number of times that the repeat can occur[6]. The optional as:setRequired="nnnn" statement controls the required occurrences that must at least be present.

`setChoice()`	`Not applicable`	
`setId()`	`as:choiceID="label"`	Associate an ID value with a part of the structure so that it can be referred to directly by ID
`setLength()`	`as:setLength="nnnn"`	Control the length of content in a structure member

[6] Design note: the setLimit / setRequired are deliberately optional. It is intended they only be used sparingly, when exceptional constraints are really needed. In W3C schema max/min are used as required factors. This impairs the ability to know when an exceptional constraint is present and therefore is an inhibitor to engineering robust interoperable systems.

Predicate	In-line Command	Notes
`setLimit()`	`as:setLimit="nnnn"`	For members that are repeatable, set a count limit to the number of times they are repeatable
`setRequired()`	`as:setRequired="nnnn"`	For members that are repeatable, set a required occurrence for the number of members that must at least be present (nnnn must be greater than 1)[7].

`setMask()`	`as:setMask=`
	`"x'Mask'"`
		Assign a regular expression or picture mask to describe the content. First character of the mask indicates the type of mask.

`setValue()`	`as:setValue="string"`	Place a value into the content of a structure

`restrictValues()`	`as:restrictValues=`
	`"[valuelist]"`
		Provide a list of allowed values for a member item

`restrictValuesByUID()`	`as:restrictValuesByUID=`
	`"UID"`
		Provide a list of allowed values for a member item from an registry reference

`useAttribute()`	`Not applicable`	
`useChoice()`	`Not applicable`	
`useElement()`	`as:useElement="true"`	Where a structure definition includes choices indicate which choice to use.
`useTree()`	`as:useTree="true"`	Where a structure member tree is optional indicate that it is to be used.
`useAttributeByID()`	`Not applicable`	
`useChoiceByID()`	`Not applicable`	
`useTreeByID()`	`Not applicable`	
`useElementByID()`	`Not applicable`	

`Not applicable`	`<include>URL`
	`</as:include>`
		Allows inclusion of an external source of assembly instructions or structure. The URL is any single valid W3C defined URL expression that resolves to physical content that can be retrieved. Note: can only be used in the <Structure> section of assembly.

[7] Design note: makeRepeatable(), makeMandatory() is the preferred syntax over the alternate: makeRepeatable() as:setRequired="1".

Predicate	In-line Command	Notes
checkCondition()	as:checkCondition= "conditionID”	Points to the condition to be tested in the data validation section.
makeRecursive()	as:makeRecursive="true"	Denotes element as a recursive structure member, so can appears as child of this parent.

The next figure 4.8.1.3 shows some examples of using these in-line commands within a structure.

Figure 4.7.1.3: Use of in-line commands with a well-formed XML structure

```
<AssemblyStructure xmlns:as="http://www.oasis-open.org/committees/cam">
 <Structure taxonomy=’XML’>
   <Items CatalogueRef=”2002”>
     <SoccerGear>
       <Item as:makeRepeatable="true">
             <RefCode as:makeMandatory="true" as:setLength="10">%%</RefCode>
             <Description>%%</Description>
             <Style>WorldCupSoccer</Style>
             <UnitPrice as:setMask="q999.9">%%</UnitPrice>
       </Item>
       <QuantityOrdered as:setMask="q999">%%</QuantityOrdered>
       <SupplierID as:makeMandatory="true">%%</SupplierID>
       <DistributorID>%%</DistributorID>
       <OrderDelivery>Normal</OrderDelivery>
       <DeliveryAddress/>
     </SoccerGear>
   </Items>
 </Structure>
</AssemblyStructure>
```

The next section shows the use of non-XML structure. It should be noted that in-line commands cannot be used with non-XML structures; all such structures require the use of predicates within the <BusinessUseContext> section of the assembly instead.

4.8.2 Non-XML structure referencing

This section shows how the CAM system supports referencing to non-XML content as shown in figure 4.8.2.1 for a legacy EDI structure definition. The XPath system can reference nodes within such structures using an appropriate node-referencing scheme that is pre-determined, (for example in an EDI transaction this would be segment identifier and field number within the segment as the node name).

Figure 4.7.2.1: An EDI example of referencing non-XML content structures

```
<AssemblyStructure xmlns:as="http://www.oasis-open.org/committees/cam">
 <Structure as:choiceID="EDI850" as:structureType="X12EDI" taxonomy=’XML’>

  <EDI Type="ASCII" Version="4040" Standard="X12">
   <TransactionSet ID="850" Name="Purchase Order" Note="">
    <Segment ID="ST" Name="Transaction Set Header" Req="M" MaxUse="1">
      <Element ID="01" Name="Transaction Set Identifier Code" Req="M"
             Type="ID" MinLength="3" MaxLength="3"
             Note="The transaction set identifier 'ST01' is used by the
             translation routines of the interchange partners to select the
             appropriate transaction set definition 'e.g., 810 select the
             Invoice Transaction Set'."/>
      <Element ID="02" Name="Transaction Set Control Number" Req="M"
             Type="AN" MinLength="4" MaxLength="9"/>
      <Element ID="03" Name="Implementation Convention Reference" Req="O"
             Type="AN" MinLength="1" MaxLength="35"
             Note="The implementation convention reference 'ST03' is used by
             the translation routines of the interchange partners to select
             the appropriate implementation convention to match the
             transaction set definition."/>
    </Segment>
    <!-- then more segments follow here... -->

   </TransactionSet>
  </EDI>
 </Structure >
</AssemblyStructure>
```

The EDI structure definition in figure 4.8.2.1 is one system for describing an EDI structure; another example would be the IGML system (http://www.igml.org) or similar systems, or a very simple system using substitution tokens as shown in figure 4.8.2.2, and then the UN/EDIFACT transaction IMPDEF system.   Another alternate to using well-formed XML as the structure example is to use a DTD or Schema instance itself.

All these may be considered for use with assembly as the business needs require.

Figure 4.7.2.2: Tokens EDI example of referencing non-XML content structures

```
<AssemblyStructure>
 <Structure as:choiceID="Healthcare Transaction" as:structureType="Tokens"
taxonomomy=’EDI’ xml:space=”preserve”>
<!-- #
ISA*00*%%*00*%%*01*%%*01*Interchange Rec*010404*1031*U*00200*000025331*0*I*:~
GS*AA*%%*%%*20010404*1031*000000000*T*004010X097~
ST*276*0001~
BHT*0010*13**%%~
HL*1**%%*1~
NM1*PR*2*%%*****PI*%%~
HL*2*1*%%*1~
NM1*41*2*%%*****46*X67E~
HL*3*2*%%*1~
NM1*1P*2*%%*****SV*987666~
HL*4*3*22*0~
DMG*D8*%%*M~
NM1*QC*1*%%*%%****MI*%%~
TRN*1*%%~
REF*BLT*%%~
AMT*T3*%%~
REF*1K*%%~
REF*BLT*%%~
AMT*T3*%%~
SE*%%*0001~
GE*1*000000000~
IEA*1*%%~
# -->
 </Structure >
</AssemblyStructure>
```

The tokens method using “%%” for the replacement items as shown in figure 4.8.2.2 is easily adapted to suit a wide variety of non-XML content structures.

An example of an XPath predicate reference would `makeRepeatable(\\HL::NM1)` for a block of lines, and `makeOptional(\\REF)` to indicate a segment line or `makeMandatory(\\AMT\01)` to indicate a field within a segment.

The comment mechanism is used to allow the EDI syntax to be placed into the XML itself, along with the XML command to preserve the white space formatting.

In each case partners using these systems must agree on the processing rules for the non-XML content they are intending to process. Industry standards bodies can also define such rules as extensions to the base CAM system for legacy payloads within their own domain. Implementers may provide a generic tokens method as a default for non-XML content since it can handle a broad range of such content.

4.8.3 Including External Structures into CAM

In the first release of CAM, the inclusion of external structure definitions is restricted to the <structure> section of the document. This ensures a reasonable level of complexity for implementations, while allowing use of existing structure definitions such as DTD or Schema specifications easily and simply. The external structure can also be a CAM aggregate component structure emitted from a modelling tool or similar means of allowing combinations of structure components together to make a complete whole. Such tools can easily use in-line commands within the structure to align the assembly process with the model definitions.

The example in figure 4.8.3.1 shows syntax for including an external structure or composite fragments of structure together for use within assembly. The business rules within the <BusinessUseContext> section can then reference these structure items to complete the functionality required.

Figure 4.7.3.1: Use of <as:include> commands within an assembly XML structure

```
<CAM CAMlevel="1" version="1.0"
     xmlns:as="http://www.oasis-open.org/committees/cam">
       <AssemblyStructure>
        <Structure taxonomy='XML'>
          <BusinessInvoice>
              <as:include>
                     http://www.oasis-open.org/strct/invoice.xml
              </as:include>
              <billingAddress>
              <as:include>
                     http://www.oasis-open.org/strct/address.xml
              </as:include>
```

```
             </billingAddress>
          </BusinessInvoice>
        </Structure>
       </AssemblyStructure>
       <BusinessUseContext/>
       <ContentReference/>
       <DataValidations/>
</CAM>
```

Include statements are assumed to retrieve consistent pieces of content, and not fragments that do not parse as a contiguous whole.

The document referenced by an <as:include> statement may contain one or more further <as:include> statements, however, if this contains a circulatory reference, then processing of the include statements should fail and stop with an appropriate error message. Nested including provides direct support for core component mechanisms and aggregate component components that can be assembled together.

Referencing into include structures using anchor references.

Since URL location references support it, an include reference may be in a format that includes reference to a standard XML element location via an Id_ref within the target structure. This would result in only that part of the structure being returned by the include. An example would be:

```
             <as:include>
                   http://www.oasis-open.org/strct/components.xml#us_address
             </as:include>
```

A similar approach can be used for HTML or other merge structure components (see use of Merge feature for more details)

.

4.8.4 Object Oriented Includes Support

In order to augment the ability of modelling tools to generate CAM structure objects, the include statement has optional parameters attached to it of extends=" " and implements=" ".

Figure 4.7.4.1 Example of CAM include with OO extensions

```
<CAM CAMlevel="1" version="1.0"
     xmlns:as="http://www.oasis-open.org/committees/cam">
       <AssemblyStructure>
        <Structure taxonomy='XML'>
          <BusinessInvoice>
              <as:include extends="SGIR:UN034500" implements="SGIR:UN034750">
                    http://www.oasis-open.org/strct/invoice.xml
              </as:include>
              <billingAddress>
              <as:include extends="SGIR:CIQ010100" implements="SGIR:CIQ010350">
                    http://www.oasis-open.org/strct/address.xml
              </as:include>
              </billingAddress>
          </BusinessInvoice>
        </Structure>
       </AssemblyStructure>
       <BusinessUseContext/>
       <ContentReference/>
       <DataValidations/>
</CAM>
```

The `extends` and `implements` parameters are optional, and the CAM processor does not parse the information contained in them. Essentially they are external notes for use in modelling tools.

Typical values may consist of a registry alias prefix with UID reference values that denote semantic content.

The next section reviews the requirements of the last step of the assembly process, which bridges to the physical business application and data content. It provides the means to formalize that step beyond the assembly and the linkage to the physical systems.

4.8.4.1 Support for import style functionality

To enhance the ability to include and re-use CAM template logic, the properties section of the CAM template has been extended to allow referencing to external CAM template logic. When using this capability, then XPath references may include an alias prefix, as in [alias::XPath] that then refers the CAM processor to explicit content in an imported CAM template for the equivalent section of the CAM template pointed to by the import reference.

Examples of this use are including sections of structure from another CAM template; referencing to BusinessContext rules from another CAM template, and DataValidation rules (note: in all such referencing this must point to a unique reference path, as the CAM processor will always return the first occurrence in the imported document that matches the path specified).

Figure 4.7.4.1.1 Example of CAM import style XPath referencing

```
<CAM CAMlevel="1" version="1.0"
     xmlns:as="http://www.oasis-open.org/committees/cam">
       <AssemblyStructure>
        <Structure taxonomy='XML'>
          <BusinessInvoice as:useTree="SGIRimport:://BusinessInvoice/Detail/">
              <billingAddress as:useTree="SGIRimport:://Address/">
              </billingAddress>
          </BusinessInvoice>
        </Structure>
       </AssemblyStructure>
       <BusinessUseContext/>
       <ContentReference/>
       <DataValidations/>
</CAM>
```

Similarly in the business use context section a constraint action can be specified that instead of specifying the behaviour – provides the import XPath expression. If there is a context condition, then the CAM processor can apply its local context values to see if any imported conditions apply, and if so, can then action any for that matching XPath expression. Any Content referencing section item references will automatically be imported and will apply, unless they are overridden by item declarations in the CAM template.

4.8.5 Merge Structure Handling and External Content Mapping

When processing a merge structure as an external mapping this requires three components of the CAM template to be used. The first two reside in the <AssemblyStructure> section and provide an input (source) and an output (target) structure layout. The merge structure itself is indicated by using the type attribute set to MERGE, and an IDreference so that the structure can be directly referenced. The third part is then provided by the <ExternalMapping> section and a cross-referencing that tallies the source field to the target field token names. Notice that the ExternalMapping section now includes Context rules also so that these can be context driven mappings. Therefore a single CAM template can produce multiple outputs as necessary, and the <location> element of the <ExternalMapping> section can be used to output to each such targets to different post-processing options.

To facilitate this functionality the following is required. An <AssemblyStructure> <Structure> section provides the Output Template File that defines the layout to be used in the operation – typically this will be a format such as HTML, xhtml or XML, but there is no restriction, except that the file contains a substitution structure of the required output. This structure will be part of the AssemblyStructure section, but with a special type of 'MERGE' to denote its use with the ExternalMapping section.

Embedded in the syntax of the substitution structure are merge tags. This works very similar to the embedded function statements already used in the <Structure> section of the <AssemblyStructure> for parsing an incoming target source structure.

[Note: this potentially means you can do a three way merge – where the input is from a structure incoming in say from a transaction in XML, the ExternalMapping refers to a SQL table in read-mode, not update-mode, and then the output structure in HTML contains references to both sets of information. (The caveat here is that there is a one-to-one correspondence between input records and the SQL table).]

Merge tags have the following generic form: { Token Prefix } { Token[_name] } { Token Suffix } where Token Prefix and Suffix are defined using attributes in the preamble of the <Structure> declaration. By default the Token Prefix and Suffix are defined as "#" and "#;" respectively. The '#' can be escaped using '\#' when output of a '#' is required.

The Token[_name] part of the tag is one of the following:

·CAM field name

#as:fieldname#; is a data tag that is matched during runtime with a data element defined in the <ExternalMapping> section of the CAM template.

If a match is found, the value replaces the tag during the <ExternalMapping> Output operation.

·as:REPEAT
The #as:REPEAT#; tag defines the beginning of a repeated area. The repeated area is duplicated and processed during each matching output operation of a repeating group within an input record

structure, thereby allowing for an unknown number of data rows.

The tag is removed from the output.

·as:ENDREPEAT

The #as:ENREPEAT; tag defines the end of a repeated area.

The tag is removed from the output.

·as:IF (expression)

The #as:IF (XPath Expression)#; tag defines the start of an IF block. The expression specified is parsed and matched against values from the input structure and or with a data element defined in the ExternalMapping section. The data is assumed to be a valid logical XPath expression and is evaluated. If the expression is True the rest of the IF block is processed. Otherwise the ELSE block is processed.

If the expression does not evaluate to a logical value, it is assumed to be False.

The tag is removed from the output.

·as:ELSE

The #as:ELSE#; tag defines the start of an ELSE block and the end of an IF block, which must precede the ELSE block. The ELSE block is processed if the XPath expression value of the IF block evaluates to False.

This tag is optional.

The tag itself is removed from the output.

·as:ENDIF

The #as:ENDIF#; tag defines the end of an IF block, or an ELSE block if one exists.

This tag is mandatory if an #as:IF(expression)#; exists.

The tag is removed from the output.

·as:INCLUDE

The #as:INCLUDE(URL)#; tag allows you to include an entire additional external file during the Merge process. This begin tag is followed by the URL of the file to be included. The file name can be a tag itself. The Include process will take place after the file is fully merged, therefore it should not contain recursive references.

Examples:

#as:INCLUDE(http:/camdemo.com/tmp/t1.html)#;

Will include the file t1.html in the current output template file.

#as:INCLUDE(#as:_T1#;)#;

Will include the file referred to by the #as:_T1#; field value in the current input record.

Syntax Rules

The number and order of the #as:REPEAT#; and #as:ENREPEAT#; tags must match.

The number and order of the #as:ELSE#;, and #as:ENDIF#; tags must match.

#as:ELSE#; tags may only be placed between a pair of #as:IF_name#; and #as:ENDIF#;.

REPEAT and IF-ELSE-ENDIF blocks can nested.

The following two figures, 4.7.5.1 and 4.7.5.2 now illustrate an example of using this functionality.

Figure 4.7.5.1: Example of mapping script to a semi-structured output format (merge)

```
<ExternalMapping>
 <ContentAssociation>
  <Description>Orders Report Monthly Detail</Description>
  <Context/>
  <InputSource type=”XML” structureID=”#myReportData”/>
  <OutputStore structureID=”#htmlReport” type=”MERGE”
location=”orders_report.html”/>
  <RulesSet>
   <MapRule output="Order Month"
       input="@STARTGRP(Sales/Company/Year/Qtr/Month)"/>
      <MapRule output="Month" input="Sales/Company/Year/Qtr/Month"/>

      <MapRule output="Order Items"
          input="@STARTGRP(/Company/Year/Qtr/Product@type)"/>
         <MapRule output="type" input="Sales/Company/Year/Qtr/Product@type"/>
         <MapRule output="name"
             input="@trim(Sales/Company/Year/Qtr/Product/Item@name)"/>
         <MapRule output="manufacturer"
             input="Sales/Company/Year/Qtr/Product/Item@manufacturer"/>
         <MapRule output="value"
             input="Sales/Company/Year/Qtr/Product/Item@value"/>
         <MapRule output="sold"
             input="Sales/Company/Year/Qtr/Product/Item@sold"/>
      <MapRule output="Order Items" input="@ENDGRP()"/>
   <MapRule output="Order Month" input="@ENDGRP()"/>
  </RulesSet>
 </ContentAssociation>
</ExternalMapping>
```

Then associated with this content mapping is the following merge structure in the <AssemblyStructure> section; note that the sequence of the @STARTGRP() statements in the external mapping section should correspond to the sequence of the #as:REPEAT#; groups in the merge target.

Figure 4.7.5.2: Merge target structure example

```
<Structure ID==”#htmlReport”>
<![CDATA[
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<!--last modified on Friday, November 21, 2003 04:20 PM -->
<HTML>
<HEAD>
 <META HTTP-EQUIV="Content-Type" CONTENT="text/html;CHARSET=iso-8859-1">
 <META NAME="GENERATOR" Content="OASIS CAM V1.0">
 <META NAME="Author" Content="OASIS CAM">
 <TITLE>Monthly Orders Report</TITLE>
</HEAD>
<BODY>
<H1><FONT COLOR="blue">Monthly Orders Report</FONT></H1>
<P>
#as:REPEAT#;
<H2><FONT FACE="Arial ">Month: #as:Month#;</FONT></H2>
<P>
<TABLE BORDER="0" WIDTH="100%">
 <TR>
  <TD WIDTH="20%" HEIGHT="42" BGCOLOR="#FFFFCC"><B><FONT
FACE="Arial">Item</FONT></B></TD>
  <TD WIDTH="20%" HEIGHT="42" BGCOLOR="#FFFFCC"><B><FONT
FACE="Arial">Type</FONT></B></TD>
  <TD WIDTH="20%" HEIGHT="42" BGCOLOR="#FFFFCC"><B><FONT
FACE="Arial">Manufacturer</FONT></B></TD>
  <TD WIDTH="20%" HEIGHT="42" BGCOLOR="#FFFFCC"><B><FONT
FACE="Arial">Units sold</FONT></B></TD>
  <TD WIDTH="20%" HEIGHT="42" BGCOLOR="#FFFFCC"><B><FONT
FACE="Arial">Value</FONT></B></TD>
 </TR>
#as:REPEAT#;
 <TR>
```

```
  <TD WIDTH="20%"><FONT FACE="Arial">#as:name#;</FONT></TD>
  <TD WIDTH="20%"><FONT FACE="Arial">#as:type#;</FONT></TD>
  <TD WIDTH="20%"><FONT FACE="Arial">#as:manufacturer#;</FONT></TD>
  <TD WIDTH="20%"><FONT FACE="Arial">#as:sold#;</FONT></TD>
  <TD WIDTH="20%"><FONT FACE="Arial">#as:value#;</FONT></TD>
 </TR>
#as:ENDREPEAT#;
#as:IF (Sales/Company/Year/Qtr/Month < "12" )#;
 </TABLE>
#as:ENDIF#;
#as:ENDREPEAT#;
 <TR>
  <TD WIDTH="20%" BGCOLOR="#FFFFCC">&nbsp;</TD>
  <TD WIDTH="20%" BGCOLOR="#FFFFCC">&nbsp;</TD>
  <TD WIDTH="20%" BGCOLOR="#FFFFCC">&nbsp;</TD>
  <TD WIDTH="20%" BGCOLOR="#FFFFCC">&nbsp;</TD>
  <TD WIDTH="20%" BGCOLOR="#FFFFCC">&nbsp;</TD>
 </TR>
</TABLE>
</BODY>
</HTML>
]]>
</Structure>
```

The section completes the processing requirements for the assembly system; the addendum now provides reference examples.

4.9 Predicate Format Options

There are several ways in which predicates can be referenced with a CAM template. The tables below show the different forms to be used and when. The first table shows the BusinessUseContext Rules format when a constraint is applying one and only one action to an element or attribute. The second table is for when a constraint is applying several actions to one item (specified by a path). The third table shows the inline functions when applied to elements. The fourth shows a proposed extension for the inline definitions to be used with attributes.

TABLE 1: Possible functions for constraint action attribute: `<as:constraint action=”`*`functiondefn`*`”/>`
`excludeAttribute(xpath)`
`excludeElement(xpath)`
`excludetree(xpath)`
`makeMandatory(xpath)`
`makeOptional(xpath)`
`makeRepeatable(xpath)`
`restrictValues(xpath,valuesList)`
`setChoice(xpath)`
`setDateMask(xpath,dateMask)`
`setID(xpath,idValue)`
`setLength(xpath,lengthDescription)`
`setLimit(xpath,limitValue)`
`setMask(xpath,datatype,Mask)`
`setValue(xpath,value)`
`useAttribute(xpath)`
`useChoice(xpath)`
`useElement(xpath)`
`useTree(xpath)`

TABLE 2: Possible function for constraint action element:
`<as:constraint item=”xpath”>`
`   <as:action>`*`functiondefn`*`</as:action>`
`</asconstraint>`

`excludeAttribute()`
`excludeElement()`
`excludetree()`
`makeMandatory()`
`makeOptional()`
`makeRepeatable()`
`restrictValues(valuesList)`
`setChoice()`
`setDateMask(dateMask)`
`setID(idValue)`
`setLength(lengthDescription)`
`setLimit(limitValue)`
`setMask(datatype,Mask)`
`setValue(value)`

useAttribute()
useChoice()
useElement()
useTree()

TABLE 3: Inline Element functions – used alongside structure example - all are attributes
as:makeMandatory="true"
as:makeOptional="true"
as:makeRepeatable="true"
as:restrictValues="valuesList" valuesList ::= value|value|... value ::= string with or without quotes
as:setChoice="idValue" all elements in choice have same idValue
as:setDateMask="dateMask"
as:setID="idValue"
as:setLength="lengthDescription" : lengthDescription = min-max or max
as:setLimit="limitValue"
as:setMask="Mask" – must be used with a as:datatype attribute for non string masks
as:setValue="value"

TABLE 4: Inline attribute functions – used alongside structure example all are attributes. Assumed to be for an attribute called 'example' - <element example="value"/>
as:makeMandatory-example="true"
as:makeOptional-example ="true"
as:restrictValues-example ="valuesList" valuesList ::= value|value|... value ::= string with or without quotes
as:setMask-example ="Mask" – must be used with a as:datatype attribute for non string masks
as:setID-example ="idValue"
as:setLength-example ="lengthDescription" : lengthDescription = min-max or max
as:setNumberMask-example ="numberMask"
as:setValue-example ="value"

4.10 Conformance Levels and Feature Sets

One goal of CAM is to provide the means for simple implementations of a limited base functionality for implementers. To facilitate this goal the implementation has been separated into three levels, where level 1 contains the minimum functionality, level 2 contains extended functionality and level 3 contains advanced features.

To aid implementers and conformance testing the following matrix shows by section those features that apply to each level. Also it should be noted that the CAM header section contains processing rules for header information relating to level control for CAM processor implementations.

Figure 4.9.1.1: CAM conformance matrix.

Feature	Document reference	Level 1	Level 2	Level 3
Header section processor		required	required	required
Structure processor, simple XML		required	required	required
Structure processor, inline predicates		partial	complete	required
Structure processor for schemas		none	none	required
Structure processor for non-XML targets		none	none	required
Include sub-assembly mechanism		partial	required	required
XPath Context rules		required	required	required
Lookup() function support		none	required	required
Reference section – local definitions		required	required	required
Reference section – external registry interfacing		Simple URL based accessing only	Simple URL and ebXML registry	complete
Validation section – simple checks		none	required	required
Validation section – extended checks		none	required	required
Validation section – external functions		none	required	required
External Mapping section		none	none	required
Picture mask support		input validate functionality only	required	required
External context system support		required	required	required
CAM import feature		none	none	required
CAM merge feature of External Mapping		none	none	required

A CAM conformance test suite will be developed and made available from the website. Also, a CAM processor when encountering CAM namespaced syntax in-line, or within the CAM template itself, that it cannot recognize should report this as a warning, and then continue to parse.

4.11 Future Feature Extensions

This section is provided as a holding area for potential extensions to the base CAM specifications.

RDF support
The ability to use RDF syntax to provide metadata and semantics in the ContentReference section for elements.

Registry based noun semantics
This is currently under development with the Registry SCM group and will be referenced here when complete.

WSDL support for CAM processor
A draft WSDL interface has been posted for discussion and is available. Implementers may use this as a basis for deploying a CAM processor as a web service.

Accessing content in ebXML Registry
The ebXML Registry Services Specification (RSS) describes this capability.
Typical functions include the QueryManager's getRegistryObject, and getRepositoryItem operations. Also there is the HTTP interface and also the SQL or Filter query interface as described by AdhocQueryRequest.

This also includes the possibility of running external library functions offered by a registry.

The registry specifications may be found at:

[3] ebXML Registry specifications
http://www.oasis-open.org/committees/regrep/documents/2.5/specs/

Import Feature
Some basic IMPORT functionality is available in this V1.0 of CAM, however this is not intended to be comprehensive or complete. Subsequent versions of CAM will enhance the basic functions available in V1.0 and allow more sophisticated sub-assembly techniques.

A Addendum

The addendum contains some sample CAM XML instances, and the formal documented schema structure for CAM. These examples are provided both in the addendum and as standalone items as separate XML instance files[8].

A1.1 Example of an Address assembly

The first example is a complete assembly bringing together the examples used in each section of this document. The focus is on the address details and the selection and control of the structure content given that address details are highly variant depending on the delivery country.

Figure A.1.1: Sample CAM template of Address content with embedded context expressions

```
<!-- Example Assembly for Address and Order items -->
<CAM CAMlevel="1" version="1.0"
     xmlns:as=http://www.oasis-open.org/committees/cam >
   <Header>
     <Description>WorldCup Soccer Order Transaction</Description>
     <Version>1.20</Version>
     <DateTime>02/12/2003</DateTime>
      <Declaration parameter='$DeliveryCountry' default='USA' datatype='string'
use='external'/>
</Header>

<AssemblyStructure >
   <Structure taxonomy='XML'>
   <Items CatalogueRef="2002">
     <SoccerGear>
       <Item as:makeRepeatable="true">
        <RefCode as:makeMandatory="true" as:setLength="10">%%</RefCode>
        <Description>%%</Description>
        <Style>WorldCupSoccer</Style>
        <UnitPrice as:setMask="q999.99">%%</UnitPrice>
       </Item>
       <QuantityOrdered as:setMask="q999">%%</QuantityOrdered>
       <SupplierID as:makeMandatory="true">%%</SupplierID>
```

[8] Implementers seeking the very latest details should reference the schema and DTD structures for CAM directly from the Internet location for developer's resources (http://cam.swiki.net) and not rely completely on the printed instance, since corrections and

```
       <DistributorID>%%</DistributorID>
       <OrderDelivery>Normal</OrderDelivery>
       <DeliveryAddress as:choiceID="USA-Street">
          <FullName>%%</FullName>
          <Street>%%</Street>
          <City>%%</City>
          <State as:setLength="2" as:makeMandatory="true">%%</State>
       </DeliveryAddress>
       <DeliveryAddress as:choiceID="USA-APObox">
          <FullName>%%</FullName>
          <APOBox>%%</APOBox>
          <City>%%</City>
          <State as:setLength="2">%%</State>
          <Country>%%</Country>
       </DeliveryAddress>

       <DeliveryAddress as:choiceID="Canada">
          <PersonName>%%</PersonName>
          <Street1>%%</Street1>
          <Street2>%%</Street2>
          <TownCity>%%</TownCity>
          <PostCode>%%</PostCode>
          <Province>%%</Province>
          <Country>Canada</Country>
       </DeliveryAddress>
     </SoccerGear>
   </Items>
 </Structure>
</AssemblyStructure>

<BusinessUseContext>
   <Rules>
    <default>
      <context> <!-- default structure constraints -->
       <constraint action="makeRepeatable(//SoccerGear)" />
       <constraint action="makeMandatory(//SoccerGear/Items/*)" />
       <constraint action="makeOptional(//Description)" />
       <constraint action="makeMandatory(//Items@CatalogueRef)" />
```

extensions to the printed formal published implementation reference documentation can lag behind. Participation in the online

```
       <constraint action="makeOptional(//DistributorID)" />
       <constraint action="makeOptional(//SoccerGear/DeliveryAddress)" />
      </context>
    </default>
    <context condition="token='//SoccerGear/SupplierID = ' and
contains(value,'SuperMaxSoccer')">
      <constraint action="makeMandatory(//SoccerGear/DeliveryAddress)"/>
    </context>
    <context condition="token='$DeliveryCountry = ' and contains(value,'USA'">
      <constraint
         action="useChoiceByID(//SoccerGear/DeliveryAddress(#USA-Street))"/>
    </context>
    <context condition="token='$DeliveryCountry' DeliveryCountry =and
contains(value ,'APO'">
      <constraint
         action="useChoiceByID(//SoccerGear/DeliveryAddress(#USA-APObox))"/>
    </context>
    <context condition="token='$DeliveryCountry '
and co=ntains(valu e,'CANADA'">
      <constraint
         action="useChoiceByID(//SoccerGear/DeliveryAddress(#Canada))"/>
    </context>
   </Rules>
</BusinessUseContext>

<ContentReference>
   <Addressing>
    <registry name="SGIR" access="registry.sgir.org:1023" method="URL"
              description="Sporting Goods Industry Registry"/>
    <registry name="SGIRWSDL" access="registry.sgir.org:1025" method="WSDL"
              description="Sporting Goods Industry Registry"/>
    <registry name="UN" access="registry.un.org:9090" method="ebXML"
              description="United Nations EDIFACT Registry"/>
    <registry name="UPS" access="registry.ups.com:7001" method="URL"
              description="United Parcels Service Registry"/>
    <registry name="USPS" access="registry.usps.gov:8080" method="URL"
              description="United States Postal Service Registry"/>
    <registry name="Local" access="rdbms.mybusiness.com:4040" method="SQL"
```

technical discussion groups is strongly recommended.

```
              description="Local Product Database stored procedures"/>
   </Addressing>

   <item type="noun" name="RefCode"
         UIDReference="SGIR010027" taxonomy="UID" registry="SGIR"/>
   <item type="noun" name="Description"
         UIDReference="SGIR010050" taxonomy="UID" registry="SGIR"/>
   <item type="noun" name="Style"
         UIDReference="SGIR010028" taxonomy="UID" registry="SGIR"/>
   <item type="noun" name="SupplierID"
         UIDReference="SGIR010029" taxonomy="UID" registry="SGIR"/>
   <item type="noun" name="CatalogueRef" UIDReference="none" taxonomy="none"
         datatype="text" setlength="4" setmask="p\d\d\d\d " />
   <item type="noun" name="DistributorID" UIDReference="none" taxonomy="none"
         datatype="text" setlength="30" />
   <item type="noun" name="UnitPrice"
         UIDReference="070010" taxonomy="EDIFACT" registry="UN"/>
   <item type="noun" name="QuantityOrdered"
         UIDReference="070011" taxonomy="EDIFACT" registry="UN"/>
   <item type="noun" name="OrderDelivery"
         UIDReference="UPS050050" taxonomy="UID" registry="UPS"/>
   <item type="defaultAssembly" name="DeliveryAddress"
         UIDReference="USPS090081:01:05" taxonomy="UID" registry="USPS"/>
 </ContentReference>
 <DataValidations>
   <Conditions
       condition="token='$DeliveryCountry' = and contains(va lue,'USA'">
       <conditional
             expression="′//UnitPrice′ and greaterthan(value,′0.00′)"
             syntax="XPath" outcome="fail"
             message="Item price not valid / missing" test="always"/>
       <conditional
             expression="′//RefCode + //UnitPrice′ and lookup(//RefCode +
             //UnitPricevalue,′SGIRWSDL:unitprice_check′)″ outcome="report"
             message="Unit price value does not match catalog" test="always"/>
       <conditional
             expression="′//SupplierID′ and
lookup(//SupplierIDvalue,′SGIRWSDL:supplierID_check′)″
             outcome="fail"
             message="Unknown Supplier ID" test="always"/>
       <conditional
              expression="’//DistributorID’ and
lookup(//DistributorIDvalue,’SGIRWSDL:distributor_check’)”
              outcome="fail"
              message="Unknown distributor ID" test="postcheck"/>
       <conditional itemRef="//QuantityOrdered"
              expression="’//QuantityOrdered’ and
lookup(//QuantityOrderedvalue,’LocalSQL:quantityOnHand()’)”
               outcome="report"
              message="Item not available / backordered" test="postcheck"/>
   </Conditions>
</DataValidations>
</CAM>
```

In this particular example the three different address formats, USA street address, USA APO box and Canadian address are selected depending on the business use context. Notice from the business perspective this effectively controls where the company will physically deliver its products.

See the main document for details on the techniques illustrated in each section of this example. The overall business capability demonstrated is the ability to use a single assembly to manage the content variants for the business process and to tie those to the context variables that determine the actual content structure for a given business scenario.

A1.2 Example of UBL Part Order OP70 and an OAGIS BOD assembly

The examples for both UBL and for the OAGIS BOD syntax (see http://www.openapplications.org) are available for download from the CAM resource site (http://www.xmlassembly.com). The OAG example is also for a parts order and shows how the base BOD mechanism expressed simply as a W3C XSD schema fails to cover the business need (see discussion in section 1 – Introduction), while the assembly for the BOD is able to provide the required business context rules and content linkage references completely.

Figure A.1.2.1: Sample of a CAM template for OAGIS BOD content

```
<CAM CAMlevel="1" version="1.0">
  <!— Download available from http://www.xmlassembly.com -->
</CAM>
```

See the main document for details on the techniques illustrated in each section of this example. The overall business capability demonstrated is the ability to use a single assembly to manage the content variants for the business process and to tie those to the context variables that determine the actual content structure for a given business scenario.

A1.3 CAM schema (W3C XSD syntax)

This section is provided for implementers wishing a formal specification of the XML structure definition for the assembly itself. However specific implementation details not captured by the XSD syntax should be referenced by studying the specification details provided in this document and clarification of particular items can be obtained by participating in the appropriate on-line e-business developer community discussion areas and from further technical bulletins supplementing the base specifications. Also a CAM template for a CAM template is being developed.

For specific details of the latest XSD documentation please see the OASIS CAM TC documents area where the latest approved XSD version is available. This is also mirrored to the open source jCAM site as well (http://jcam.org.uk).

See document download area from OASIS website : http://www.oasis-open.org/committees/cam

A1.4 Business Process Mechanism (BPM) Context Support

This section provides an overview of the mechanism for providing context variables between the CAM processor and the remainder of the eBusiness architecture stack (see figure 2.7.2).

The CAM template provides the %parameter% mechanism to accept values from external processes. However the need is to provide a consistent mechanism in XML syntax for the propagation and specifying of context variables and their values throughout the components that make up the architecture stack.

Figure A1.4.1 shows a basic XML structure for carrying such values and it is anticipated that further development of this will continue with other OASIS TC groups to reach agreement on exact details of this mechanism. Also support for the UBL / CCTS specialized context mechanism is inherent in this generalized mechanism, and an example of such context use is also provided here, see figure A1.5.3 below.

When an <ebContext> structure is associated directly with a CAM template it can rely on the content referencing and data typing from the template to direct parsing of conditions. However, to facilitate standalone use of the <ebContext> structures, re-use can be made of CAM functions in conjunction with the xmlns:as namespace declaration. Most conditions are anticipated to be denominated lists, so the as:member() function can be used for that. Alternatively for string values such as part numbers, as:setLength() and as:setMask() can be used to specify the data constraints, while standard data types can be used for numeric and date values.
Some condition examples are shown in Figure A1.4.1 and these equate to the conceptual semantic model using parameters for category, classification, industry, type and language labelling.

This approach provides a lightweight implementation, while stopping short of requiring a complete CAM template to describe the ebContext structure itself. Instead a subset of the CAM features should be adequate for the anticipated constrained use cases of context documents (see Figure A1.4.2 below).

Figure A1.4.1 XML structure for eBusiness context variable exchange.

```
<ebContext UIDref='SDIR03400' interchangeID='123456789' BPMref='ABC123456:01'
CPAref='ABC012345'
xmlns:as="http://www.oasis-open.org/committees/cam">
<header>
 <description>An example context instance</description>
 <version>1.0</version>
 <language refcode='eng' codelist='ISO639-2' name='English'/>
 <usage>CAM</usage>
 <usage>BPM</usage>
</header>
<conditions>
   <condition name="Country"     value="USA" as:member="USA,CA,MX"
                                 as:context="GP"/>
   <condition name="itemType"    value="nonperishable" label="Item type:"
             as:member="nonperishable,perishable,refridgerated,fragile,heavy"
             as:context="PC"/>
   <condition name="partnerType" value="wholesale" label="Partner type:"
             as:member="wholesale,retail,distributor,oem,service"/>
   <condition name="Catalogue" value="A2003-Q1" as:setLength="8"
             as:setMask="sXNNNN-QN" as:UIDreference="SGIR:030451"/>
</conditions>
</ebContext>
```

Figure A1.4.2. Table of CAM features subset for ebContext usage

Function name	Required?
member()	Yes
setLength()	Yes
setMask()	Yes
UIDreference()	Optional
datatype()	Yes

This table contains a suggested selection of functions that will provide the bulk of typical functionality in configuring context instances. Notice that implementers may also choose to allow additional functions to be inserted as annotations that are simply ignored by the processor, but will act of notes for reference.

The UBL / CCTS mechanism further categorizes context variables using the following classifications.

Figure A1.4.3 UBL / CCTS context classifications

Business process (BP)	Process, collaboration, or transaction.
Business process role (BPR)	Sender and receiver roles.
Supporting role (SR)	Third party supporting role.
Industry classification (IC)	Vertical industry sector
Product classification (PC)	Type of product or service
Geopolitical (GP)	Trading region
Official constraints (OC)	Legal or contractual requirements
System capabilities (SC)	Restrictions of physical system or compliance constraints.

The examples previously provided give constraint examples in the area of geopolitical and supporting role contexts. The use of the optional in-line attribute `as:context` allows provision for use of this classification of context.

A1.5 CAM Processor Notes (Non-Normative)

CAM processor notes assist implementers developing assembly software, these are non-normative. Within an assembly implementation the processor examines the assembly document, interprets the instructions, and provides the completed content structure details given a particular set of business context parameters as input. This content structure could be stored as an XML DOM structure for XML based content, or can be stored in some other in-memory structure format for non-XML content. Additionally the memory structure could be temporarily stored and then passed to a business application step for final processing of the business content within the transaction.

Since typical development environments already contain linkage between the XML parser, the DOM, an XPath processor, a scripting language such as JavaScript, the data binding toolset such as XSLT or a comparable mapping tool. The assembly approach based on an XML script fits naturally into this environment.

Some suggested uses and behaviours for CAM processors include:

- Design time gathering of document parts to build a context sensitive assembly service that can be called via an API or webservice interface.

- Design time generation of validation scripts and schemas for the run time environment that is not CAM savvy or that does not provide any context flexibility. Think of this as a CAM compiler. This would mean that context parameters would be passed in as input to this.

- Runtime validation engine based on context parameters and controlled via a Business Process engine with BPM script definitions running within the gateways of trading partners.

Processing modes and sequencing

Context elements can have conditions. These conditions can either be evaluated against variables (parameters) or XPath statements. These conditions can be evaluated in two modes:

1) A standalone CAM template - i.e. on the basis of external parameters values passed to the CAM processor to evaluate the conditionals.
2) CAM template and XML instance - check the XML instance to evaluate the condition and then proceed (this is the normal mode for a CAM processor).

The first mode is typically used when you are trying to produce documentation about what is allowed for a transaction and it is useful to pre-process (precompile) the structure rules without the existence of an XML instance file. This means that any condition that falls into the second category can not be evaluated (these conditions then behave equivalent of having Schematron asserts, and are documented but not actioned).

A1.6 Deprecated DTD

Figure 1.6.1: Deprecated CAM structure definition in DTD syntax – this is provided for reference only, and is not being maintained.

```
<!-- CAM structure for OASIS CAM. February 10th, thru February 2004
This DTD structure is provided for reference only as it is more compact to read
and comprehend; the schema definition is for preferred normative use.

Modification history:
1.0  Initial
Revision 0.12
Revision 0.13
Revision 0.14
Revision 0.15
Copyright (c) 2003/2004 OASIS. All rights reserved.
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that such redistributions retain this
copyright notice.
This CAM structure is provided "as is" and there are no expressed or implied
warranties. In no event shall OASIS be liable for any damages arising out of
the use of this structure.  -->

<!ELEMENT CAM (Header, AssemblyStructure, BusinessUseContext?,
ContentReference?, DataValidations?, ExternalMapping?) >
<!ATTLIST CAM
          CAMlevel (1 | 2 | 3) #REQUIRED
          version (CDATA) #IMPLIED >

<!ELEMENT AssemblyStructure (Structure+)>
<!ELEMENT Header (Description?, Owner?, Version?, DateTime, ContextStatements?,
Properties?)>
<!ELEMENT Description (#PCDATA) >
<!ELEMENT Owner       (#PCDATA) >
<!ELEMENT Version (#PCDATA) >
<!ELEMENT DateTime (#PCDATA) >

<!ELEMENT ContextStatements ( ContextURL?, Declaration*)>
<!ELEMENT ContextURL (#PCDATA) >
<!ELEMENT Declaration EMPTY >
```

```
<!ATTLIST Declaration
          name CDATA #REQUIRED
          values    CDATA #IMPLIED
          default   CDATA #IMPLIED
          datatype  CDATA #IMPLIED
          use ( local | global | override ) #REQUIRED >

<!ELEMENT Structure ANY >
<!ATTLIST Structure
          ID         CDATA    #IMPLIED
          reference  CDATA    #IMPLIED
          taxonomy (XSD | DTD | RNG | XML | EDI | HTML | MERGE | OTHER)
#REQUIRED >

<!ELEMENT BusinessUseContext (Rules)>
<!ELEMENT Rules (default?, context*)>
<!ELEMENT default (context+ | constraint+ )>
<!ELEMENT context (context+ | constraint+ )>
<!ATTLIST context
        condition CDATA #REQUIRED >

<!ELEMENT ContentReference (Addressing,item*)>
<!ELEMENT Addressing (registry+)>

<!ELEMENT constraint (action+) >
<!ATTLIST constraint
        condition CDATA #IMPLIED
        action CDATA #REQUIRED >
<!-- predicates (  excludeAttribute | excludeElement | excludeTree |
                  makeOptional | makeMandatory | makeRepeatable |
                  setChoice | setId | setLength | setLimit | setMask |
                  setValue | restrictValues | restrictValuesByUID |
                  useAttribute | useChoice | useElement | useTree |
                  useAttributeByID | useChoiceByID | useElementByID |
                  useTreeByID ) -->

<!ELEMENT DataValidations ( Conditions+ )>
<!ELEMENT Conditions ( conditional+ )>
<!ATTLIST Conditions
          conditionID      CDATA #IMPLIED
          condition        CDATA #IMPLIED >
<!ELEMENT conditional EMPTY >
<!ATTLIST conditional
          expression CDATA #REQUIRED
          syntax (XPath | JavaScript | VB | Perl | Other) #IMPLIED
          outcome ( fail | ignore | report ) #REQUIRED
          message CDATA #IMPLIED
          test    ( always | postcheck | precheck )  #REQUIRED >

<!ELEMENT registry EMPTY>
<!ATTLIST registry
          name CDATA #REQUIRED
          access CDATA #REQUIRED
          method (URL | http | SOAP | ebXML | UDDI | Other) #REQUIRED
          description CDATA #IMPLIED
>

<!ELEMENT item EMPTY>
<!ATTLIST item
        type (noun | corecomponent | BIE | aggregate | defaultAssembly |
identifier | verb | schema | documentation) #REQUIRED
        name CDATA #IMPLIED
        UIDReference CDATA #REQUIRED
        taxonomy CDATA #REQUIRED
        registry CDATA #IMPLIED
         datatype CDATA #IMPLIED
         setlength CDATA #IMPLIED
         setmask CDATA #IMPLIED
>

<!ELEMENT ExternalMapping (ContentAssociation+) >
<!ELEMENT ContentAssociation
(Description?,Context,InputSource,OutputChoice,RulesSet) >
<!ELEMENT InputSource EMPTY >
<!ATTLIST InputSource
           structureID CDATA #IMPLIED
          type ( SQL | XML | EDI | TXT | ODBC | OTHER ) #IMPLIED
          location CDATA #IMPLIED >
<!ELEMENT OutputChoice (OutputStore+)>
<!ELEMENT OutputStore EMPTY >
```

```
<!ATTLIST OutputStore
          structureID CDATA #IMPLIED
          type ( SQL | XML | EDI | TXT | ODBC | XHTML | XFORM| MERGE | OTHER )
#IMPLIED
          location CDATA #IMPLIED >

<!ELEMENT RulesSet (MapRule+) >
<!ELEMENT MapRule EMPTY >
<!ATTLIST MapRule
          output CDATA #REQUIRED
          input CDATA #REQUIRED >
<!ELEMENT properties (annotation?, using?) >
<!ELEMENT using (use+) >
<!ELEMENT use (CAMlocationURL, relatedStructureID?, Description?, import+) >
<!ELEMENT CAMlocationURL (#PCDATA) >
<!ELEMENT relatedStructureID (#PCDATA) >
<!ELEMENT import EMPTY >
<!ATTLIST import
          CAMmember   CDATA #REQUIRED
          CAMalias    CDATA #REQUIRED
          comment     CDATA #IMPLIED >

<!ELEMENT annotation  (documentation+) >

<!ELEMENT documentation  (#PCDATA) >

<!ATTLIST documentation
               type (description | note | license | usage | other)   #REQUIRED
>
```

5 References

- XML Path Language (XPath) specifications document, version 1.0, W3C Recommendation 16 November 1999, http://www.w3.org/TR/xpath/

- Extensible Markup Language (XML) specifications document, version 1.1, W3C Candidate Recommendation, 15 October 2002, http://www.w3.org/TR/xml11/

- XNL: Specifications and Description Document, OASIS CIQ TC, http://www.oasis-open.org/committees/ciq

- XAL: Specifications and Description Document, OASIS CIQ TC, http://www.oasis-open.org/committees/ciq

- ISO 16642 – Representing data categories http://www.loria.fr/projets/TMF/

- CEFACT – Core components specifications - http://webster.disa.org/cefact-groups/tmg/

- UN – eDocs resource site - http://www.unece.org/etrades/unedocs/

- UN – Codelists reference site for eDocs - http://www.unece.org/etrades/unedocs/codelist.htm

- Jaxen reference site - http://jaxen.org/