OASIS Content Assembly Mechanism Specification Version 1.1

Committee Specification 01

10 April 2007

Specification URIs:

This Version:

http://docs.oasis-open.org/cam/v1.1/cs01/OASIS-CAM-Specification-1_1-015-041007.html
http://docs.oasis-open.org/cam/v1.1/cs01/OASIS-CAM-Specification-1_1-015-041007.doc
http://docs.oasis-open.org/cam/v1.1/cs01/OASIS-CAM-Specification-1_1-015-041007.pdf

Previous Version:

http://docs.oasis-open.org/cam/v1.1/cd02/OASIS-CAM-Specification-1_1-RC-015-031407.html
http://docs.oasis-open.org/cam/v1.1/cd02/OASIS-CAM-Specification-1_1-RC-015-031407.doc
http://docs.oasis-open.org/cam/v1.1/cd02/OASIS-CAM-Specification-1_1-RC-015-031407.pdf

Latest Version:

http://docs.oasis-open.org/cam/v1.1/OASIS-CAM-specification-v1_1.html
http://docs.oasis-open.org/cam/v1.1/OASIS-CAM-specification-v1_1.doc
http://docs.oasis-open.org/cam/v1.1/OASIS-CAM-specification-v1_1.pdf

Latest Approved Version:

http://docs.oasis-open.org/cam/OASIS-CAM-specification.html
http://docs.oasis-open.org/cam/OASIS-CAM-specification.doc
http://docs.oasis-open.org/cam/OASIS-CAM-specification.pdf

Technical Committee:

OASIS Content Assembly Mechanism TC

Chair(s):

David RR Webber

Editor(s):

Martin Roberts
David RR Webber

Related work:

This specification replaces or supercedes:

- OASIS CAM v1.0 committee draft

This specification is related to:

- OASIS ebXML specifications (ISO 15000)
- OASIS web services specifications
- W3C XPath, namespaces, XSD and XML specifications

Declared XML Namespace(s):

xmlns:as=http://docs.oasis-open.org/cam/xmlns
asm1, asm2, asm3, default namespaces placeholders as needed

Abstract:

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.

Status:

This document was last revised or approved by the Content Assembly Mechanism TC on the above date. The level of approval is also listed above. Check the "Latest Version" or "Latest Approved Version" location noted above for possible later revisions of this document.

Technical Committee members should send comments on this specification to the Technical Committee's email list. Others should send comments to the Technical Committee by using the "Send A Comment" button on the Technical Committee's web page at http://www.oasis-open.org/committees/cam/

The CAM TC work is operating on an open license approach charter with unencumbered content, see the Technical Committee web page at http://www.oasis-open.org/committees/cam/charter.php

For information relating to disclosure of patents pertaining to the CAM TC work, and if any such contributing member statements exist, please refer to the Intellectual Property Rights section of the Technical Committee web page (http://www.oasis-open.org/committees/cam/ipr.php)

The non-normative errata page for this specification is located at http://www.oasis-open.org/committees/cam/

Notices

Table of Contents

Figures and Tables

1 Introduction

The core role of CAM remains the same - defining, composing and validating XML content. The version 1.1 of the CAM specification seeks to simplify the original work and more clearly delimit between core normative features and extended non-normative sections and items. Also V1.1 builds from lessons learned over the past two years in developing actual CAM templates. The new approach aligns closely with common industry practice in marshalling and unmarshalling XML content, the XML DOM and allows the use of common XML tools, including rule engines, alongside the CAM toolset. Consequently the CAM toolset now provides a powerful set of typical XML scripted functional components that by default are needed when exchanging XML business transactions.

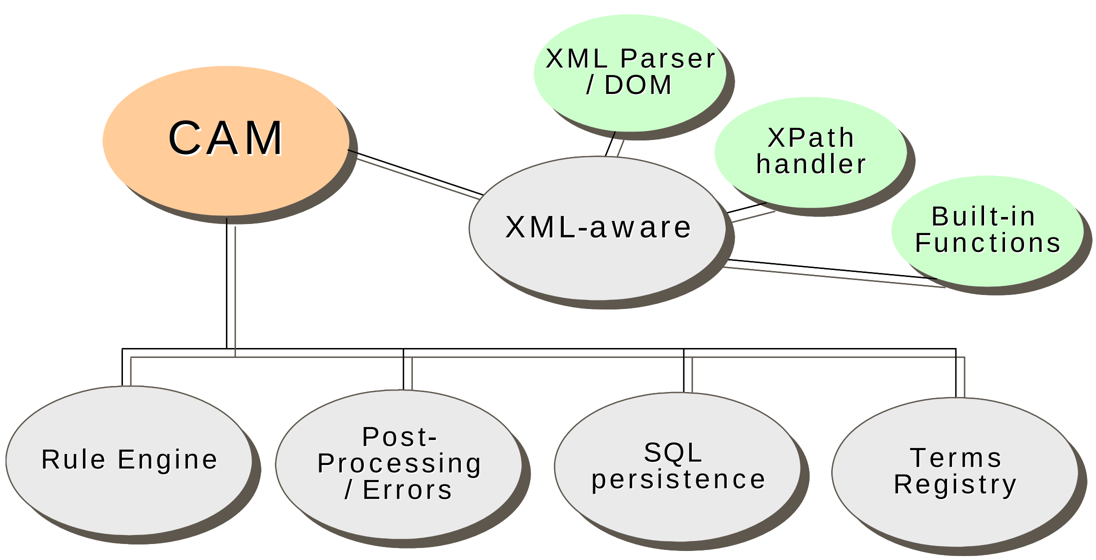
The XML scripting is designed to be obvious, human readable and declarative. This means that the task of providing rule-driven control mechanisms can become open and re-usable across an ebusiness community of practice, not just for localized internal point solutions. This is especially important in today's web service environments to support the concept of loose-coupling of service interfaces and their associated transaction interchanges. We have also taken into account the W3C and OMG work on rules.

The objective in releasing v1.1 is to provide a foundation specification that is simple, clear and easy to implement today. Whereas the new approach now allows integration with specialized tools that link into backend database systems and/or handles specialized structure formats, specialized error handling mechanisms or provide engines for complex rule based logic. In addition support for external context mechanisms are provided to align with business process needs, such as the OASIS ebBP/BPSS.

This approach is designed to separate the common sharable needs from the in-house local specializations in a coherent systematic way. This allows implementers to isolate their own point development and still align with common community practice and core business information handling structures and rules.

Future extensions to the specification may then build out and provide additional normative tools as extended areas are better formalized and common industry practice establishes itself. An example of the need to develop further normalized specification parts include registry interfacing and marshalling and unmarshalling to and from SQL content repositories. Today these are provided by specialized tools and CAM provides a formal extension mechanism and application programming interface (API) for these non-normative needs.

Figure 1 - The implementation model for a CAM processor



Referencing Figure 1 - the top-most XML-aware functions are normative components required of a CAM processor to support the core XML-scripting functionality. The lower components are optional tools supported by the pluggable interface that CAM v1.1 provides. Implementers can use local specialized tools as determined by their specific application environment. It is envisioned this implementation model can be developed using a variety of modern programming languages and the pluggable interface is supported by tools such as the Apache Foundation Maven technology. This flexibility allows for support of W3C Rule Interchange Format (RIF) and OMG Production Rule Representation (PRR) as applicable.

1.1 Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in **RFC2119** (see abbreviation references below).

All text is normative unless otherwise labelled.

1.2 Normative 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/

- XML Schema Definitions (XSD) **– [XSD1]** XML Schema Part 1: Structures, W3C Recommendation 2 May 2001 http://www.w3.org/TR/xmlschema-1/

  http://www.oasis-open.org/committees/download.php/6248/xsd1.html

  [XSD2] XML Schema Part 2: Datatypes, W3C Recommendation 2 May 2001

  http://www.w3.org/TR/xmlschema-2/

  http://www.oasis-open.org/committees/download.php/6247/xsd2.html

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

- XAL: Specifications & 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/

1.3 Non-Normative References

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

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

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

1.4 Terms and Definitions

Assembly model

A tree-structured model that can be implemented as a document schema.

Class diagram

A graphical notation used by [UML] to describe the static structure of a system, including object classes and their attributes and associations.

Component model

A representation of normalized data components describing a potential network of associations and roles between object classes.

Context

The circumstance or events that form the environment within which something exists or takes place.

Dependency diagram

A refinement of a class diagram that emphasizes the dependent associations between object classes.

Document

A set of information components that are interchanged as part of a business transaction; for example, in placing an order.

Functional dependency

A means of aggregating components based on whether the values of a set of properties change when another set of properties changes, that is, whether the former is dependent on the latter.

Normalization

A formal technique for identifying and defining functional dependencies.

Spreadsheet model

A representation of an assembly model in tabular form.

XSD schema

An XML document definition conforming to the W3C XML Schema language [XSD1][XSD2].

The terms Core Component (CC), Basic Core Component (BCC), Aggregate Core Component (ACC), Association Core Component (ASCC), Business Information Entity (BIE), Basic Business Information Entity (BBIE), and Aggregate Business Information Entity (ABIE) if used in this specification refer to the meanings given in [CCTS].

The terms Object Class, Property Term, Representation Term, and Qualifier are used in this specification with the meanings given in [ISO11179].

The keywords MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY and OPTIONAL, when they appear in this document, are to be interpreted as described in [RFC2119].

1.5 Symbols and Abbreviations

ABIE

Aggregate Business Information Entity

ACC

Aggregate Core Component

ASBIE

Association Business Information Entity

ASCC

Association Core Component

ASN.1

ITU-T X.680-X.683: Abstract Syntax Notation One; ITU-T X.690-X.693: ASN.1 encoding rules

http://www.itu.int/ITU-T/studygroups/com17/languages/X.680-X.693-0207w.zip

http://www.oasis-open.org/committees/download.php/6320/X.680-X.693-0207w.zip

BBIE

Basic Business Information Entity

BCC

Basic Core Component

BIE

Business Information Entity

CC

Core Component

CCTS

UN/CEFACT ebXML Core Components Technical Specification 2.01

http://www.untmg.org/downloads/General/approved/CEFACT-CCTS-Version-2pt01.zip

http://www.oasis-open.org/committees/download.php/6232/CEFACT-CCTS-Version-2pt01.zip

EAN

European Article Numbering Association

EDI

Electronic Data Interchange

ISO

International Organization for Standardization

ISO11179

ISO/IEC 11179-1:1999 Information technology — Specification and standardization of data elements — Part 1: Framework for the specification and standardization of data elements

http://www.iso.org/iso/en/ittf/PubliclyAvailableStandards/c002349_ISO_IEC_11179-1_1999(E).zip

http://www.oasis-open.org/committees/download.php/6233/c002349_ISO_IEC_11179-1_1999%28E%29.pdf

JSDF

Java Simple Date Format library

NDR

UBL Naming and Design Rules (see Appendix B.4)

RFC2119

Key words for use in **RFC**s to Indicate Requirement Levels

http://www.faqs.org/rfcs/rfc2119.html

http://www.oasis-open.org/committees/download.php/6244/rfc2119.txt.pdf

S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, http://www.ietf.org/rfc/rfc2119.txt, IETF RFC 2119, March 1997.

UML

Unified Modeling Language [UML] Version 1.5 (formal/03-03-01)

http://www.omg.org/docs/formal/03-03-01.pdf

http://www.oasis-open.org/committees/download.php/6240/03-03-01.zip

UN/CEFACT

United Nations Centre for Trade Facilitation and Electronic Business

XML

Extensible Markup Language [XML] 1.0 (Second Edition),W3C Recommendation 6 October 2000

http://www.w3.org/TR/2000/REC-xml-20001006

http://www.oasis-open.org/committees/download.php/6241/REC-xml-20001006.pdf

XSD

W3C XML Schema Language [XSD1] [XSD2]

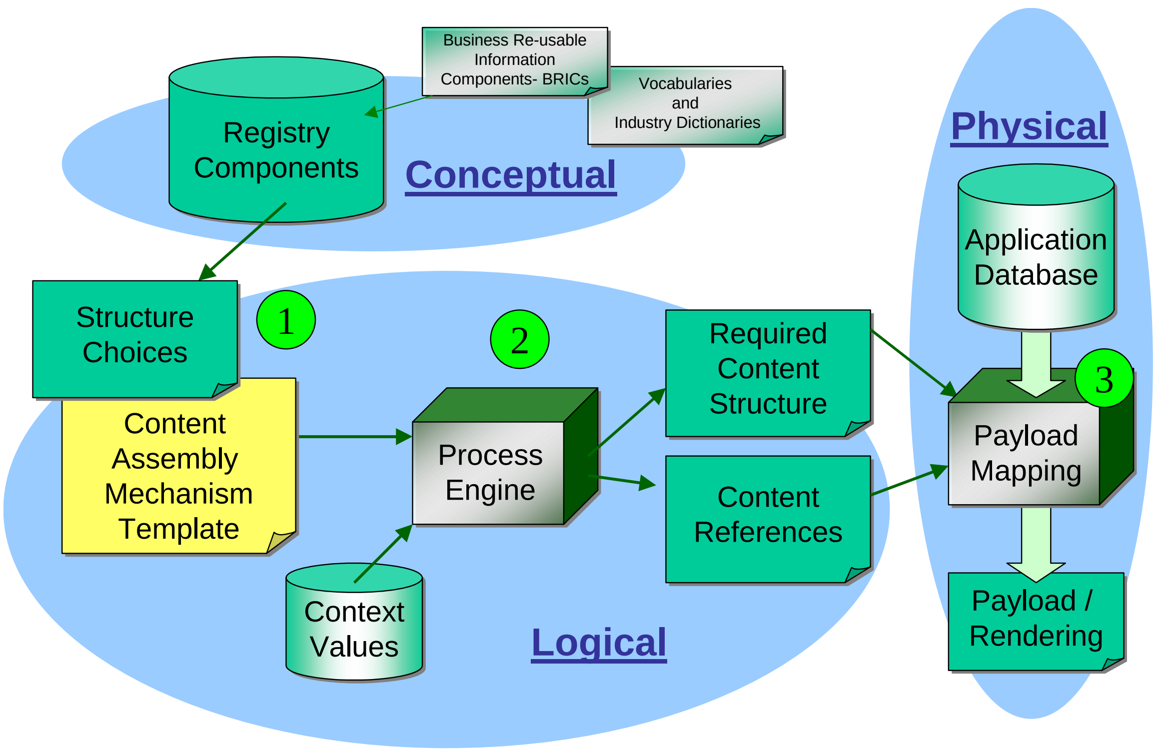
2 Pre-requisites

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.

3 Content Assembly Mechanism Technical Specification

This section describes the implementation specifications for CAM. As noted above there are three roles to CAM – defining, composing and validating content. Figure 1 shows how implementers can integrate CAM technology into their existing content generation systems, while Figure 2 shows CAM in a content validation role, and then Figure 3 shows defining content rules.

Figure 2 - Deploying CAM Technology – Context Driven Assembly



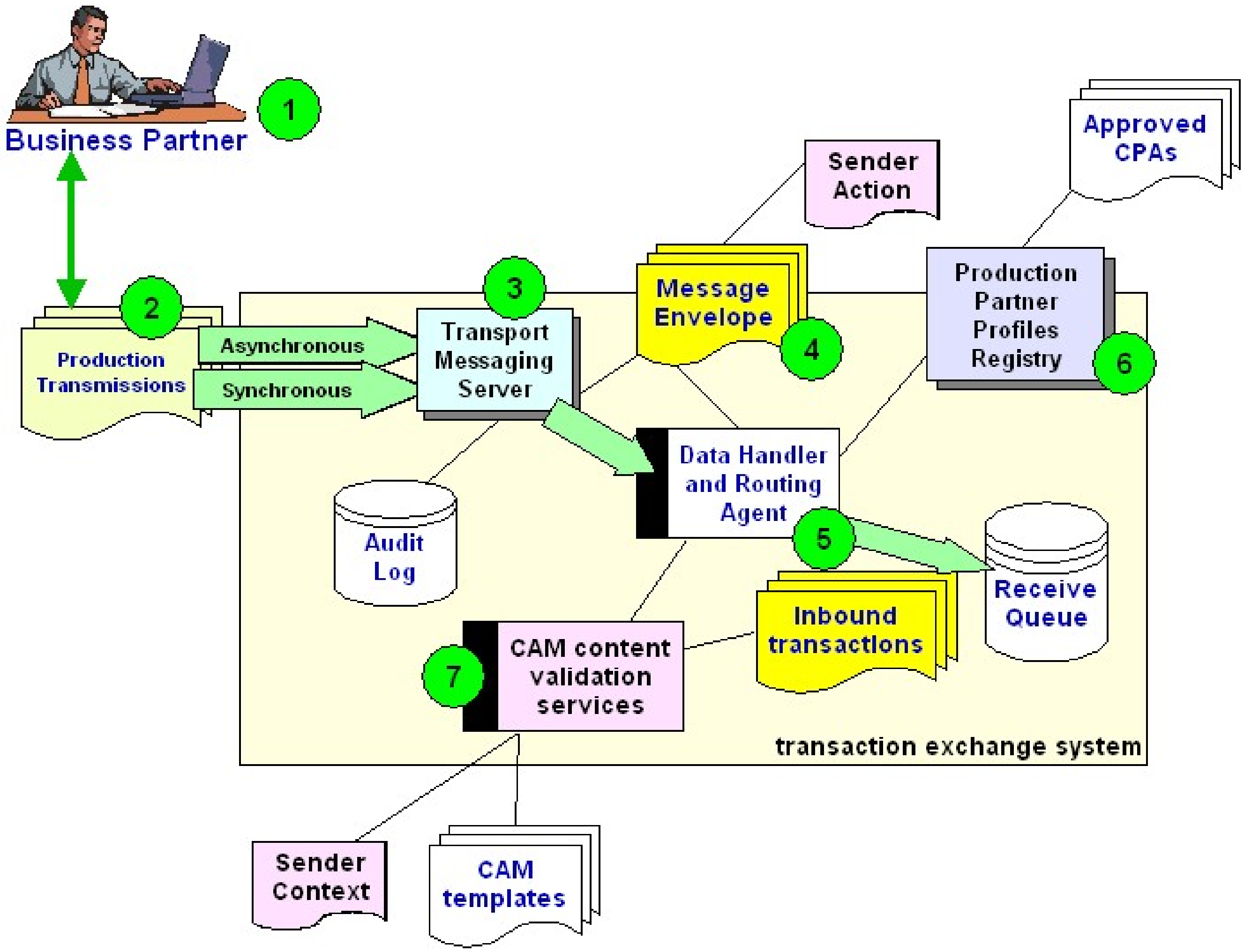
In reference to Figure 2 - Deploying CAM Technology – Context Driven Assembly, item 1 is the subject of this section, describing the syntax and mechanisms. Item 2 is a process engine designed to implement the CAM logic as an executable software component, and similarly item 3 is the application XML marshalling and unmarshalling component that links the e-business software to the physical business application software and produces the resultant transaction payload for the business process needs.

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 using a fully-featured eBusiness deployment environment example.

The Figure 2 shows how CAM can be integrated as a content validation service within a transactional exchange system using partner profiles, context and actions to drive transaction validation.

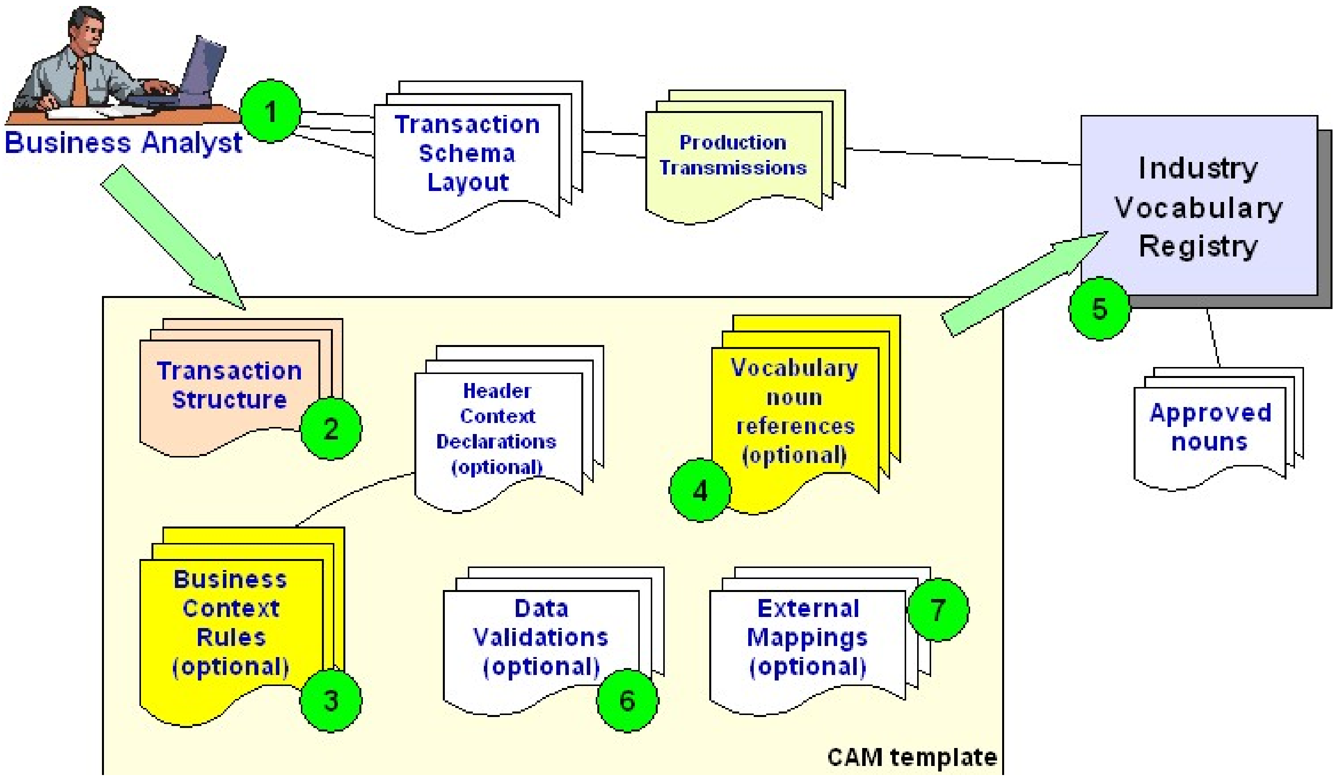
Figure 3 - Deploying CAM technology – Context Driven Validation



Referencing the Figure 3 - Deploying CAM technology – Context Driven Validation, the business partner (#1) sends business transactions (#2) to the partners messaging server (#3). The messaging envelope (#4) contains the sender action and the data handler (#5) checks that against the partner profiles on record in the Registry (#6). The sender action from the envelope also determines via the CPA (Collaboration Partner Agreement) the CAM template associated with that business process step. The data handler (#5) then invokes the CAM validation services (#7) and passes the references to: the inbound transaction on the receive queue, the sender context and the CAM template. The CAM validation services (#7) then verifies the content and returns either the precise error details found or a valid transaction status back to the data handler for action. Using this configuration allows CAM to act as a context driven validation service that is configurable via the partner CPA, the Sender Action from the message envelope received, and the CAM templates defined for the business process.

Then Figure 4 below provides a lower level of detail into the XML script mechanisms required and the business analysis steps that lead to the definition of these contents.

Figure 4 – Deploying CAM technology – Defining Content Rules and Structures



Referencing Figure 4 above the business analyst examines the business transaction schema layouts (#1), the sample production transmissions, and references the industry vocabulary dictionary. Using the CAM template the actual transaction structure required (#2) is defined. This may optionally contain additional context rules (#3) that direct CAM processing based on variables and values (the header section can contain global context declarations). Then noun references may also be created (#4) that cross-reference between the structure elements (#2) and the registry dictionary (#5) and the approved industry noun definitions. Optionally local application validation rules (#6) may also be added that test specific local requirements and also optional (#7) is the application mappings (such as database table columns). Used in this role the CAM template captures the information exchange details in an XML template that can then be shared and referenced between partners and agreed to as the business information requirements.

The tools from both Figure 3 and Figure 4 can also be deployed interactively via a web browser interface to allow partners to pre-test, and / or, self-certify prior to production message exchanges being sent. This can provide online interactive tools where sample XML transactions can be tested by upload to a CAM validation tool that applies the selected template and reports online any errors detected.

3.1 Overview

The CAM itself consists of four logical sections and the CAM template is expressed in XML syntax. This is shown in figure 5 as high-level XML structure parent elements[1].

Figure 5 - High-level parent elements of CAM (in simple XML syntax)

```
<CAM CAMlevel="1" version="1.1">
  <Header>
  <AssemblyStructure/>
  <BusinessUseContext/>
  <Extension/> <!—Optional, repeatable -->
</CAM>
```

The structure sections provide the core of the publically agreed interchange definition between exchange partners - *A*ssembly Structure(s), and *B*usiness Use Context Rules. Then the internal pre- or post processing can be referenced as local include extensions as needed for specializations.

The optional extensions and includes are envisioned to support specialized non-normative handling that in the prior CAM specification functionality included items such as *C*ontent References (with optional associated data validation), extended *D*ata Validations including rule agents and marshalling/unmarshalling content via *E*xternal Mappings. These process needs are now retained as future potential normative items that are still evolving and described in a non-normative companion document to the main V1.1 specification as Appendix B.

Figure 6 - Structure for entire CAM syntax at a glance[2] next shows the complete v1.1 specification hierarchy for CAM at a glance.

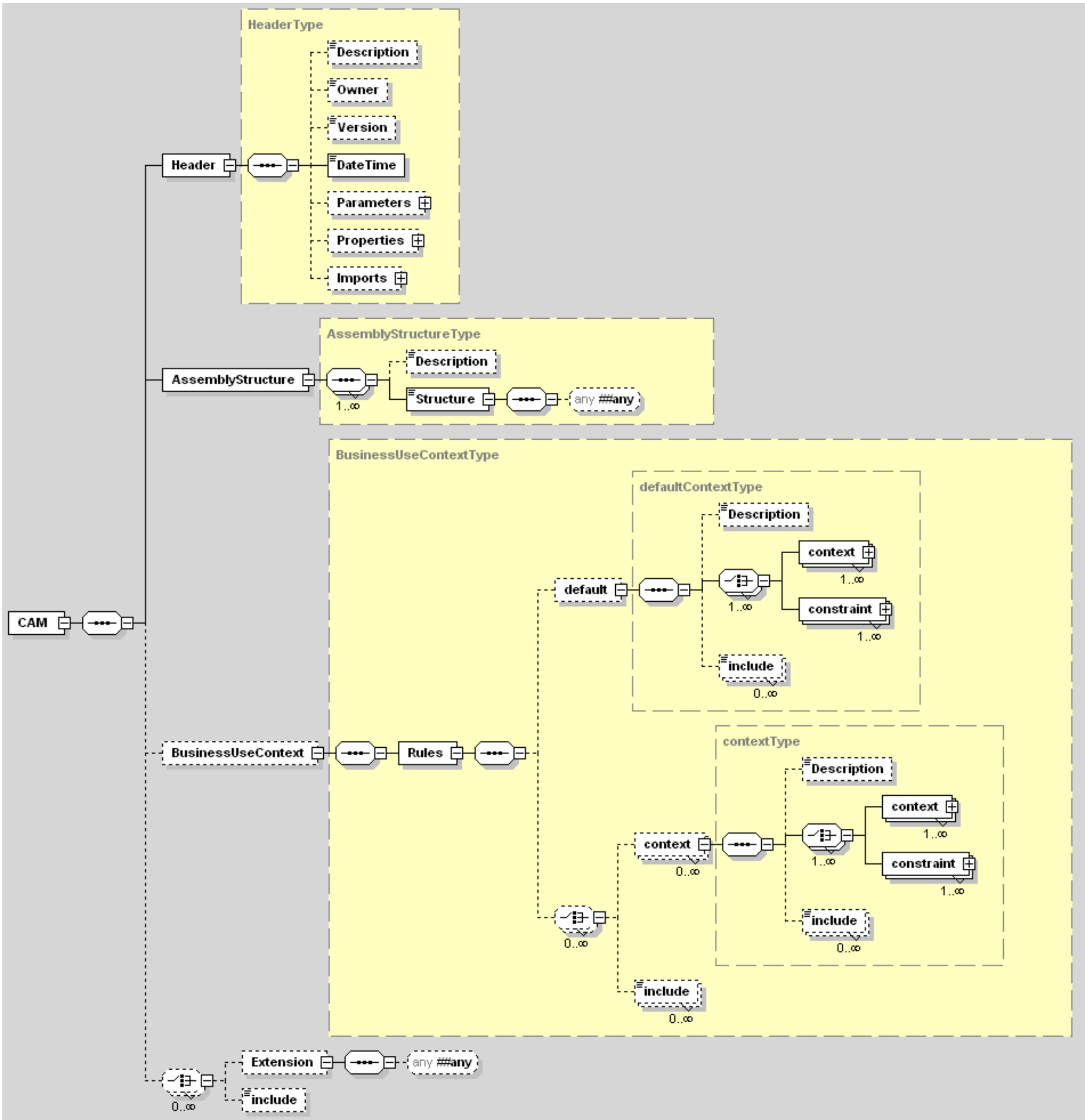
The CAM header it should be noted has built-in support for compatibility levels within the specification to both aid in implementation of the CAM tools, and also to ensure interoperability across versions.

This is controlled via the CAMlevel attribute of the CAM root element. More details on the CAM implementation levels and features are provided in advanced options section later.

[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 6 - Structure for entire CAM syntax at a glance



Each of the parent items is now described in detail in the following sub-sections, while the formal schema definition for CAM is provided at the OASIS web site in machine readable Schema format XSD syntax. While the documented schema provides a useful structural overview, implementers should always check for the very latest version on-line at the docs.oasis-open.org/cam area 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.

3.2 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.

3.2.1 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. CAM uses the $name syntax to denote external parameter references where required in the CAM template statements. External parameters can be passed using the CAM context mechanism (see later section on Advanced Features support).

3.2.2 Pseudo Variables

This item is non-normative, level 2.

When processing documents it is often expedient to have access to the system time. This would allow checks against that time to be made and therefore validation to check for example that delivery dates are in the future. To do this CAM defines the following pseudo variables.

- $date – this gives today's date in the format YYYY-MM-DD
- $time – this gives the time at the start of processing the incoming file in the format HH:MI:SS
- $dateTime – this is a combination of the previous variables in the format YYYY-MM-DDTHH:MI:SS

These variables should be set by the processor at the start of processing for each incoming document.

In addition there is a need for date math functions to be provided to allow checks against the current time and date and also between date fields. The following is considered a minimal set that may be provided.

These functions compare a field with the date or time of the validation:

- dateAfterNow(xpath,dateMask)
- timeAfterNow(xpath,timeMask)
- dateBeforeNow(xpath,dateMask)
- timeBeforeNow(xpath,timeMask)

The following functions allow either a positive or negative integer, which represents either days or hours to be added to Now:

- dateAfterDays(xpath,dateMask,numofdays)
- timeAfterHours(xpath,dateMask,numofhours)
- dateBeforeDays(xpath,dateMask,numofdays)
- timeBeforeHours(xpath,dateMask,numofhours)

The following functions allow comparison between two fields:

- after(xpath,mask,xpath,mask)
- before(xpath,mask,xpath,mask)

3.2.3 Properties

This item is non-normative, level 2.

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.

3.2.4 Imports

This item is non-normative, level 2.

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 include statements would have the format:

```
<as:include>$importname/xpath</as:include>
```

An example with an import declared as ‘common_rules’ would be as follows:

```
<as:include>$common_rules//as:BusinessUseContext/as:Rules/as:default</as:include>
```

This example will load any default rules from the ‘common_rules’ CAM Template into the current template.

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

3.3 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. The current V1.x series of the specification uses simple well-formed XML throughout to illustrate the usage. 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[3], 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).

[3] EDI is used in the generic sense through out this document to refer to the family of pre-XML text markup systems, such as EDI-X12, UN/EDIFACT, HL7, FIX, SWIFT and more. See http://www.disa.org for more details on EDI technologies. Each flavour of EDI can be accommodated within the AssemblyStructure section of the CAM template as needed.

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 0 below. The Figure 7 – Example of Structure and format for AssemblyStructure here shows a simple example for an AssemblyStructure using a single structure for content.

Figure 7 – Example of Structure and format for AssemblyStructure

```
<Header>
     <Description>Example 4.2.1 using structures</Description>
     <Version>0.05</Version>
</Header>
<AssemblyStructure>
 <Structure taxonomy=”XML”> //XML is the only allowed value for Version 1.1
<!-- 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 8
-->
 </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 and use of include directives. These can also be included from external sources, with nesting of assemblies; see the section below on Advanced Features for details. Also a mechanism is provided to select a structure based on an XPath reference to content within an XML instance.

To provide the direct means to express content values within the structure syntax the following two methods apply. A variable substitution value for an element or attribute is indicated by text that must start and end with a ‘%’sign, for example ‘%Description%’; or simply %% where no indicative content is preferred. Any other value is assumed to be a fixed content value. Figure 8 - Substitution and fixed parameters values, with a well-formed XML structure shows examples of this technique.

Figure 8 - Substitution and fixed parameters values, with a well-formed XML structure

```
<Header>
      <Description>Example 4.2.2 Well-formed XML structure</Description>
      <Version>1.0</Version>
   <as:Parameters>
          <as:Parameter name="DeliveryCountry"
 values="USA|Mexico|Canada|Europe "
 use="Global"
 default="USA"/>
   </as:Parameters>

</Header>
<AssemblyStructure>
 <Structure taxonomy=”XML” ID=”SoccerGear”>
   <Items CatalogueRef=”2006”> //Fixed Value
<SoccerGear>
   <Item>
<RefCode>%000_00_0000%</RefCode> // Value subject to rules
<Description>%any text line%</Description>
<Style>WorldCupSoccer</Style>
<UnitPrice>%amount%</UnitPrice>
   </Item>
<QuantityOrdered>%integer%</QuantityOrdered>
<SupplierID>%%</SupplierID>
<DistributorID>%%</DistributorID>
<OrderDelivery>Normal</OrderDelivery>
<DeliveryAddress>
   <USA>        // details of address here
```

```
    </USA>
    <Mexico>     // details of address here
    </Mexico>
  <Canada>        // details of address here
  </Canada>
  <Europe>        // details of address here
  </Europe>
</DeliveryAddress>
</SoccerGear>
    </Items>
  </Structure>
</AssemblyStructure>
```

Referring to Figure 8 - Substitution and fixed parameters values, with a well-formed XML structure, the “2006”, “WorldCupSoccer” and “Normal” are fixed values that will always appear in the payload transaction at the completion of the CAM processing of the content.

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.

3.4 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).

Note: By default CAM assumes that any XML structure item, element or attribute, is mandatory unless a rule is added in the BusinessUseContext section or an inline rule is placed in the structure.

Note: By default CAM will not enforce order of elements within an XML structure unless a rule is added in the BusinessUseContext section or an inline rule is placed in the structure (same behaviour as with XML 1.0 attributes ordering being undetermined). This feature makes CAM templates more flexible, particularly for complex structures, and prevents erroneous error flagging.

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 (aka occurrence usage rules) of the structure members and content definitions. Figure 9 - The Assertion predicates for BusinessUseContext shows the structure assertion predicates.

Figure 9 - The Assertion predicates for BusinessUseContext

excludeAttribute()	**useAttribute()**
excludeElement()	**useChoice()**
excludeTree()	**useElement()**
makeOptional()	**useTree()**
makeMandatory()	**useAttributeByID()**
makeRepeatable()	**useChoiceByID()**
setChoice()	**useElementByID()**
setId()	**useTreeByID()**
setLength()	**startBlock()**
setLimit()	**endBlock()**
setValue()	**checkCondition()**
setDateMask()	**makeRecursive()**
setStringMask()	**setUID()**
setNumberMask()	**restrictValues()**
datatype() or setDataType()	**restrictValuesByUID()**
setRequired()	**orderChildren()**
allowNulls()	**setDefault()**
	setNumberRange()

Each predicate provides the ability to control the cardinality of elements[4] 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 non-normative 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 are applied 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. BusinessUseContext default rules and predicates.
3. BusinessUseContext conditional rules and predicates.

Referring to the structure in the example shown in Figure 8 - Substitution and fixed parameters values, with a well-formed XML structure, Figure 10 – Syntax example for BusinessUseContext 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 a variable that has been created from the Parameter or the Properties section in the Header.

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.

[4] Predicates can also be used on attributes as well as elements in the XML structure.

Figure 10 – 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 using parameter DeliveryCountry-->
     <constraint action="useTree(//SoccerGear/DeliveryAddress/USA)"/>
   </context>
</Rules>
</BusinessUseContext>
```

Referring to the XPath expressions in Figure 10 – Syntax example for BusinessUseContext, 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 11 - Matrix of predicates for BusinessUseContext declarations) 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.

494

Figure 11 - 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, [minvalue-maxvalue]**	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
setDateMask() **setStringMask()** **setNumberMask()**	**//memberpath, [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** **setDatatype()**	**//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.

Predicate	Parameter(s)	Description
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)[5].
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).
as:datetime() **Non-Normative,level 2**	**date-picture-mask** **date-picture-mask + P7D** **date-picture-mask – P30D**	Allows variables to contain computed date values for use in rule comparisons or setting event timings (value is returned from system clock of server)
setUID() **Non-Normative,level 2**	**//memberpath, alias, value**	Assign a UID value to a structure element. Alias must be declared in registry addressing section of ContentReferences).
restrictValues()	**//memberpath,** **[valuelist],[defaultValue]**	Provide a list of allowed values for a member item
restrictValuesByUID()	**//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).

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

Predicate	Parameter(s)	Description
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() **Non-Normative**	**StructureID**	As per useAttribute but referenced by structure ID defined by SetId or in-line ID assignment
useChoiceByID() **Non-Normative**	**StructureID**	As per useChoice but referenced by structure ID defined by SetId or in-line ID assignment
useTreeByID() **Non-Normative**	**StructureID**	As per useTree but referenced by structure ID defined by SetId or in-line ID assignment
useElementByID() **Non-Normative**	**StructureID**	As per useElement but referenced by structure ID defined by SetId or in-line ID assignment
checkCondition() **Non-Normative,level 2**	**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. Note that if the orderChildren() is set the recursive element must occur after all the other children.
startBlock() **Non-Normative,level 2**	**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()` **`Non-Normative,level 2`**	**`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.
`orderChildren()`	**`//elementpath`**	This means that the children must occur within the element in the order that they occur in the Structure provided. This overrides the default CAM behaviour which is to allow child elements to occur in any order.
`allowNulls()`	**`//memberpath`**	When used for elements either the XML empty syntax <empty/> format or the <empty></empty> format would be accepted as valid mandatory content. For attributes they are permitted to be empty i.e. no white space or any characters between value delimiters ("" or ‘’). *Note: This is to enable a similar functionality to the "nillable" function in xsd, however the user would not have to supply the XML instance xsi:nil="true" attribute.*
`setDefault()`	**`//memberpath`**	Sets the default value for a node to the value given (applies to element or attribute) when the item is empty or missing (if optional). This will allow defaults to be applied either directly or in conjunction with the restrictValues() function. *Note: This can also apply with the lookup() extension function (non-normative).*

Predicate	Parameter(s)	Description
setNumberRange()	**//memberpath**	For use with nodes of content type number. This would allow the specification of a number being between two values inclusively (e.g. 0-10 would include 0 and 10). *Note: This supplements the restrictValues() function for nodes of type number.*

The predicates shown in Figure 11 - Matrix of predicates for BusinessUseContext declarations) 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.

3.4.1 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 12 - XPath Comparator functions below extend this to provide the complete set of familiar logical comparisons.

Figure 12 - 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 > value**	Conditionally check for a greater value
Less than	**value < value or value < value**	Conditionally check for a lesser value
Greater than or equal	**value >= value or value >= value**	Conditionally check for a greater than or equal to value
Less than or equal	**Value <=value or value <= 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')`	This 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.
`after`	`after(xpath, DateMaskPicture,$pseudovariable)`	Non-normative extra function for comparison of dates and times
`before`	`before(xpath, DateMaskPicture,$pseudovariable)`	Non-normative extra function for comparison of dates and times

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.

3.4.2 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. This specification does not stipulate how to treat CDATA sections.

3.4.3 CAM content 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, CAM provides 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 below and usage details can be found by studying the examples provided in the example tables.

The ability to support alternate date mask syntax for dates, such as with the Java Simple Date and Numeric Format (JSDF / JSNF) syntax[6] and class methods, is now also permitted and a mechanism described.

The JSDF / JSNF functionality is very similar to the original CAM mask system but provides some extra capabilities and formats.

(Note: this technique can allow use of alternate mask systems syntaxes such as SQL, Perl, and so on as may be required for specific industry / partner use).

Description

Picture masks are categorized by the basic data-typing element that they can be used in combination with. CAM processors must check the content of the element or attribute against the masks and report any errors.

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

String Pictures

The positional directives and mask characters for string pictures are as follows:

X - any character mandatory

Aa - A for alphanumeric mandatory and a for alphanumeric optional may include spaces

? – 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

' ' – single quotes, escape character block to denote actual mandatory character

Examples of string pictures are shown in the following table:

[6] See details of SDF at - http://java.sun.com/j2se/1.4.2/docs/api/java/text/SimpleDateFormat.html

String value	Picture mask (shorthand)	Full expanded mask	Validation match
portability	X6	XXXXXX	portab
portability	UX3	UXXX	Port
portability	XXXXing	XXXXing	porting
realtime	XXXX-XXXX	XXXX-XXXX	real-time
BOLD!	L5	LLLLL	bold!
asX	XX'X'	XX'X'	Matches asX but not asd

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

J - Uppercase, first character of – invoke alternate optional Java character format library methods to handle mask processing – character J is ignored in actual mask (see alternate masks item below)

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

Numeric value	Picture
-1234.56	######.##
-1234.56	000000.##
-1234.56	-######.##
0	-######.##Z* where Z indicates zero suppress – e.g. 000000.01 becomes 0.01

Basic 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).

J - Uppercase, first character of – invoke alternate optional Java character format library methods to handle mask processing – character J is ignored in actual mask (see alternate masks item below)

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	**Validation Matches**
MM/DD/YYYY	03/21/1992
MMMMMMMMMM^DDDD, ^YYYY	March^^^^^^21st,^1992
MMMMMMMMMM^DDDD, ^YYYYT	March^21st,^1992 with trimming directive (see below)
WWWWWWWWWW^-^W	Saturday^^^-^7
WWWWWWWWWW^-^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 MMMMMMMMMM,^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 '-'

3.4.3.1 Alternate Simple Date Format - Date and Time Patterns

The simple date and time formats are specified by *date and time pattern* strings[7]. Within date and time pattern strings, unquoted letters from `'A'` to `'Z'` and from `'a'` to `'z'` are interpreted as pattern letters representing the components of a date or time string. Text can be quoted using single quotes (`'`) to avoid interpretation, where `"''"` represents a single quote. All other characters are not interpreted; they're simply copied into the output string during formatting or matched against the input string during parsing.

The following tables provide details of the patterns and their usage.

A compliant implementation should first check the initial character of the picture mask. If it is uppercase J character – then the mask is assumed to be of Java simple format. Then the processor should pass the mask to the equivalent alternate mask processor – such as the Java Simple Date Format method - for either date or time handling, and if that then fails – then an error should be returned.

The following pattern letters are defined (all other characters from `'A'` to `'Z'` and from `'a'` to `'z'` are reserved):

Letter	Date or Time Component	Presentation	Examples
`G`	Era designator	Text	`AD`
`y`	Year	Year	`1996`; `96`
`M`	Month in year	Month	`July`; `Jul`; `07`
`w`	Week in year	Number	`27`
`W`	Week in month	Number	`2`
`D`	Day in year	Number	`189`
`d`	Day in month	Number	`10`

[7] Source: Sun Java documentation - http://java.sun.com/j2se/1.4.2/docs/api/java/text/SimpleDateFormat.html

F	Day of week in month	Number	`2`
E	Day in week	Text	`Tuesday`; `Tue`
a	Am/pm marker	Text	`PM`
H	Hour in day (0-23)	Number	`0`
k	Hour in day (1-24)	Number	`24`
K	Hour in am/pm (0-11)	Number	`0`
h	Hour in am/pm (1-12)	Number	`12`
m	Minute in hour	Number	`30`
s	Second in minute	Number	`55`
S	Millisecond	Number	`978`
z	Time zone	General time zone	`Pacific Standard Time`; `PST`; `GMT-08:00`
Z	Time zone	RFC 822 time zone	`-0800`

Pattern letters are usually repeated, as their number determines the exact presentation:

- **Text:** For formatting, if the number of pattern letters is 4 or more, the full form is used; otherwise a short or abbreviated form is used if available. For parsing, both forms are accepted, independent of the number of pattern letters.
- **Number:** For formatting, the number of pattern letters is the minimum number of digits, and shorter numbers are zero-padded to this amount. For parsing, the number of pattern letters is ignored unless it's needed to separate two adjacent fields.
- **Year:** For formatting, if the number of pattern letters is 2, the year is truncated to 2 digits; otherwise it is interpreted as a number.

For parsing, if the number of pattern letters is more than 2, the year is interpreted literally, regardless of the number of digits. So using the pattern "MM/dd/yyyy", "01/11/12" parses to Jan 11, 12 A.D.

For parsing with the abbreviated year pattern ("y" or "yy"), `SimpleDateFormat` must interpret the abbreviated year relative to some century. It does this by adjusting dates to be within 80 years before and 20 years after the time the `SimpleDateFormat` instance is created. For example, using a pattern of "MM/dd/yy" and a `SimpleDateFormat` instance created on Jan 1, 1997, the string "01/11/12" would be interpreted as Jan 11, 2012 while the string "05/04/64" would be interpreted as May 4, 1964. During parsing, only strings consisting of exactly two digits, as defined by `Character.isDigit(char)`, will be parsed into the default century. Any other numeric string, such as a one digit string, a three or more digit string, or a two digit string that isn't all digits (for example, "-1"), is interpreted literally. So "01/02/3" or "01/02/003" are parsed, using the same pattern, as Jan 2, 3 AD. Likewise, "01/02/-3" is parsed as Jan 2, 4 BC.

- **Month:** If the number of pattern letters is 3 or more, the month is interpreted as text; otherwise, it is interpreted as a number.
- **General time zone:** Time zones are interpreted as text if they have names. For time zones representing a GMT offset value, the following syntax is used:
- *GMTOffsetTimeZone:*
- `GMT` *Sign Hours* `:` *Minutes*
- *Sign:* one of

- `+ -`
- *Hours:*
  - *Digit*
  - *Digit Digit*
- *Minutes:*
  - *Digit Digit*
- *Digit:* one of

  `0 1 2 3 4 5 6 7 8 9`

Hours must be between 0 and 23, and *Minutes* must be between 00 and 59. The format is locale independent and digits must be taken from the Basic Latin block of the Unicode standard.

For parsing, RFC 822 time zones are also accepted.

- **RFC 822 time zone:** For formatting, the RFC 822 4-digit time zone format is used:
- *RFC822TimeZone:*
  - *Sign TwoDigitHours Minutes*
- *TwoDigitHours:*

  Digit Digit

TwoDigitHours must be between 00 and 23. Other definitions are as for general time zones.

For parsing, general time zones are also accepted.

`SimpleDateFormat` also supports *localized date and time pattern* strings. In these strings, the pattern letters described above may be replaced with other, locale dependent, pattern letters. `SimpleDateFormat` does not deal with the localization of text other than the pattern letters; that's up to the client of the class.

3.4.3.2 Examples

The following examples show how date and time patterns are interpreted in the U.S. locale. The given date and time are 2001-07-04 12:08:56 local time in the U.S. Pacific Time time zone.

Date and Time Pattern	Examples
`"yyyy.MM.dd G 'at' HH:mm:ss z"`	`2001.07.04 AD at 12:08:56 PDT`
`"EEE, MMM d, ''yy"`	`Wed, Jul 4, '01`
`"h:mm a"`	`12:08 PM`
`"hh 'o''clock' a, zzzz"`	`12 o'clock PM, Pacific Daylight Time`
`"K:mm a, z"`	`0:08 PM, PDT`
`"yyyyy.MMMMM.dd GGG hh:mm aaa"`	`02001.July.04 AD 12:08 PM`
`"EEE, d MMM yyyy HH:mm:ss Z"`	`Wed, 4 Jul 2001 12:08:56 -0700`

"yyMMddHHmmssZ"	010704120856-0700

3.4.3.3 Alternate Simple Decimal Format - Number Patterns

The Java simple decimal formats are specified by patterns that represent the number formatting required[8]. These patterns are selected using an uppercase J character to indicate the pattern syntax.

3.4.3.4 Patterns

`DecimalFormat` patterns have the following syntax:

Pattern:

    PositivePattern

    PositivePattern ; *NegativePattern*

PositivePattern:

    Prefix$_{opt}$ *Number Suffix*$_{opt}$

NegativePattern:

    Prefix$_{opt}$ *Number Suffix*$_{opt}$

Prefix:

    any Unicode characters except \uFFFE, \uFFFF, and special characters

Suffix:

    any Unicode characters except \uFFFE, \uFFFF, and special characters

Number:

    Integer Exponent$_{opt}$

    Integer . *Fraction Exponent*$_{opt}$

Integer:

    MinimumInteger

    #

    # *Integer*

    # , *Integer*

MinimumInteger:

    0

[8] Java simple decimal format - http://java.sun.com/j2se/1.4.2/docs/api/java/text/DecimalFormat.html

0 *MinimumInteger*
0 , *MinimumInteger*

Fraction:
MinimumFraction$_{opt}$ *OptionalFraction*$_{opt}$

MinimumFraction:
0 *MinimumFraction*$_{opt}$

OptionalFraction:
OptionalFraction$_{opt}$

Exponent:
E *MinimumExponent*

MinimumExponent:
0 *MinimumExponent*$_{opt}$

A `DecimalFormat` pattern contains a positive and negative subpattern, for example, `"#,##0.00;(#,##0.00)"`. Each subpattern has a prefix, numeric part, and suffix. The negative subpattern is optional; if absent, then the positive subpattern prefixed with the localized minus sign (code>'-' in most locales) is used as the negative subpattern. That is, `"0.00"` alone is equivalent to `"0.00;-0.00"`. If there is an explicit negative subpattern, it serves only to specify the negative prefix and suffix; the number of digits, minimal digits, and other characteristics are all the same as the positive pattern. That means that `"#,##0.0#;(#)"` produces precisely the same behavior as `"#,##0.0#;(#,##0.0#)"`.

The prefixes, suffixes, and various symbols used for infinity, digits, thousands separators, decimal separators, etc. may be set to arbitrary values, and they will appear properly during formatting. However, care must be taken that the symbols and strings do not conflict, or parsing will be unreliable. For example, either the positive and negative prefixes or the suffixes must be distinct for `DecimalFormat.parse()` to be able to distinguish positive from negative values. (If they are identical, then `DecimalFormat` will behave as if no negative subpattern was specified.) Another example is that the decimal separator and thousands separator should be distinct characters, or parsing will be impossible.

The grouping separator is commonly used for thousands, but in some countries it separates ten-thousands. The grouping size is a constant number of digits between the grouping characters, such as 3 for 100,000,000 or 4 for 1,0000,0000. If you supply a pattern with multiple grouping characters, the interval between the last one and the end of the integer is the one that is used. So `"#,##,###,####"` == `"######,####"` == `"##,####,####"`.

3.4.3.5 Special Pattern Characters

Many characters in a pattern are taken literally; they are matched during parsing and output unchanged during formatting. Special characters, on the other hand, stand for other characters, strings, or classes of characters. They must be quoted, unless noted otherwise, if they are to appear in the prefix or suffix as literals.

The characters listed here are used in non-localized patterns. Localized patterns use the corresponding characters taken from this formatter's `DecimalFormatSymbols` object instead, and these characters lose their special status. Two exceptions are the currency sign and quote, which are not localized.

Symbol	Location	Localized?	Meaning
`0`	Number	Yes	Digit

`#`	Number	Yes	Digit, zero shows as absent
`.`	Number	Yes	Decimal separator or monetary decimal separator
`-`	Number	Yes	Minus sign
`,`	Number	Yes	Grouping separator
`E`	Number	Yes	Separates mantissa and exponent in scientific notation. *Need not be quoted in prefix or suffix.*
`;`	Subpattern boundary	Yes	Separates positive and negative subpatterns
`%`	Prefix or suffix	Yes	Multiply by 100 and show as percentage
`\u2030`	Prefix or suffix	Yes	Multiply by 1000 and show as per mille
`¤` (`\u00A4`)	Prefix or suffix	No	Currency sign, replaced by currency symbol. If doubled, replaced by international currency symbol. If present in a pattern, the monetary decimal separator is used instead of the decimal separator.
`'`	Prefix or suffix	No	Used to quote special characters in a prefix or suffix, for example, `"'#'#"` formats 123 to `"#123"`. To create a single quote itself, use two in a row: `"# o''clock"`.

For more information, examples and pattern manipulation see the documentation for the Java DecimalFormat method and links to examples there. The library also supports use of scientific notation numbers.

3.5 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: Functions used for constraint action attribute: <as:constraint action="*functiondefn*"/>
excludeAttribute(xpath)
excludeElement(xpath)
excludetree(xpath)
makeMandatory(xpath)
makeOptional(xpath)
makeRepeatable(xpath)

TABLE 1: Functions used for constraint action attribute: `<as:constraint action="functiondefn"/>`
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)
orderChildren(xpath)

TABLE 2: Function used 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()
orderChildren()

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 single 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"
as:orderChildren=”true”

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"

3.6 In-line use of predicates and references

Figure 8 in Section 3.3 above shows an example for an AssemblyStructure with different structure components for address (e.g. US, Europe, Canada). Using different structures for content can be controlled with in-line statements indicating by context those optional and required 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 for setChoice() and useChoice(). The table in Figure 13 below gives the list of these in-line statements and the equivalent predicate form where applicable.

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. For attributes inline functions can be included by using the format **‘as:attributename-functionname=”value”’**. .

The in-line statements available are detailed in the table shown in Figure 13. 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. See Figure 14 below for examples of using in-line predicates.

Figure 13 - Matrix of in-line statement commands and predicate commands

Predicate	In-line Command	Notes
excludeAttribute()	**Not applicable**	
excludeElement()	**Not applicable**	
excludeTree()	**Not applicable**	

Predicate	In-line Command	Notes
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; leaf element may not be nillable
allowNull()	**as:allowNull="true"**	Allow null content model for leaf element
makeRepeatable()	**as:makeRepeatable="true"** **as:setLimit="5n"** **as:setRequired="3n"**	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[9]. 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-mmmm"**	Control the length of content in a structure member
setLimit()	**as:setLimit="nnnn"**	For members that are repeatable, set a count limit to the number of times they are repeatable

[9] 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
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)[10].
setDateMask() **setNumberMask()** **setStringMask()**	**as:setDateMask="DD-MM-YY"** **as:setNumberMask="####.##"** **as:setStringMask="U8"** **"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="'value'|'value'"** **"[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**	

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

Predicate	In-line Command	Notes
Not applicable	**<as:include>URL </as:include>** **<as:include ignoreRoot=”yes”>**	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. The optional ignoreRoot attribute permits inclusion of fragments of XML that are not well-formed by ignoring the root element from the XML source content.
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.
orderChildren()	**as:orderChildren=”true”**	Denotes that the children of the element must occur in the order they occur in the reference structure template.

The next Figure 14 shows some examples of using these in-line commands within a structure.

Figure 14 - 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:setNumberMask="q999.9###.##">%%</UnitPrice>
     </Item>
<QuantityOrdered as:setNumberMask="q999####">%%</QuantityOrdered>
<SupplierID as:makeMandatory="true">%%</SupplierID>
<DistributorID>%%</DistributorID>
<OrderDelivery>Normal</OrderDelivery>
<DeliveryAddress/>
     </SoccerGear>
   </Items>
 </Structure>
</AssemblyStructure>
```

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.

3.7 Advanced Features

The following sections contain advanced feature options and use details.

3.8 Use of namespace declarations

The default CAM template assumes that all that is required is one namespace declaration for use with in-line CAM predicates within a template (e.g. <myTagName as:setValue=”xxx”>).

However many business vocabularies have adopted wholesale use of namespace prefixes for the elements and attributes in their schemas regardless of whether this is necessary or not. While this is not an issue for the design of CAM it is an issue for several of the XML parser implementations and the way they have been coded, including their DOM representations. Essentially when multi-namespace declarations exist in an XML instance they can no long support the default namespace having no prefix.

Unfortunately this is a common behaviour that has been widely copied due to sharing of the underlying Java libraries involved. Another issue is the placing of namespace declarations. Again the XML specifications permit these to occur anywhere in the XML instance. However the Java library implementation will often fail if all namespace declarations are not placed at the top of the XML instance.

To resolve this CAM templates permit the use of a global namespace at the root CAM template level and placing all namespace declarations in the root element declaration. You should only need to resort to this when handling structures that involve multiple inline namespace declarations within the XMl content. Processors can provide a function to extract namespace definitions from an XML example and correctly define a CAM template skeleton with namespaces moved to the root node and any anonymous namespaces provided with a prefix (the jCAM editor implementation provides an example of this, along with the autogenerate template feature in jCAM itself). The figure 15 here illustrates an example.

Figure 15 - An example of namespace declarations for CAM templates

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Sample CAM Template showing use of namespaces extensions -->
<as:CAM CAMlevel="1" version="0.13"
  xmlns:as="http://www.oasis-open.org/committees/cam"
  xmlns:tic="http://era.nih.gov/Projectmgmt/SBIR/CGAP/ticket.namespace"
  xmlns:cb="http://era.nih.gov/Messaging/SBIR/CGAP/ticket.namespace"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.oasis-open.org/committees/cam
file:///D:/eclipse/workspace/camprocessor/schema/CAMv0151.xsd">
 <!-- note: namespace declarations should all be here, not in body of CAM
template -->
<as:Header>
<as:Description>Validates an Incoming transaction</as:Description>
<as:Owner>CAM smaple templates</as:Owner>
<as:Version>0.1</as:Version>
<as:DateTime>2004-09-09T17:00:00</as:DateTime>
<as:Parameters>
<!-- example parameter declaration -->
<as:Parameter name="applicationType" values="competing_continuation|80|70"
use="global" default="competing_continuation"/>
</as:Parameters>
</as:Header>
<as:AssemblyStructure>
<as:Structure ID="default" taxonomy="XML">
<cb:MessageType>
  <tic:ticket>
   <tic:institutionID>%%</tic:institutionID>
   <tic:correctionID>%%</tic:correctionID>
   <tic:timestamp>%%</tic:timestamp>
   <tic:application>
    <tic:projectTitle>%text%</tic:projectTitle>
    <tic:applicationType>%%</tic:applicationType>
    <tic:revisionNumber>%%</tic:revisionNumber>
```

```
    </tic:application>
   </tic:ticket>
</cb:MessageType>
</as:Structure>
</as:AssemblyStructure>
<as:BusinessUseContext>
<as:Rules>
<as:default>
<as:context>
<as:constraint action="setNumberMask(//tic:institutionID,#9)"/>
<as:constraint action="restrictValues(//tic:correctionID,'N'|'Y')"/>
<as:constraint action="setDateMask(//tic:timestamp,YYYY-MM-DDTHH:MI:SS)"/>
<as:constraint
action="restrictValues(//tic:applicationType,'competing_continuation'|'other')"/>
<as:constraint action="setNumberMask(//tic:revisionNumber,##)"/>
</as:context>
</as:default>
<!-- example additional rules -->
<as:context>
</as:context>
</as:Rules>
</as:BusinessUseContext>
</as:CAM>
```

3.9 Extending CAM Processors

Originally CAM v1.0 was designed to have 5 distinct areas within the template. These were to cover off expected forms of content handling and advanced functionality. In the 1.1 specification these have been replaced in favour of a more extensible framework. This framework is based on the idea of a CAM processor being able to provide a core set of XML handling functions, while allowing extensions via the optional include or ANY functionality.

An extension entry is designed to allow CAM processors to invoke functionality that is too specialized to allow strict normative definition by the CAM specification and implementation by the CAM processor developers (such as for local integration specialization needs, error handling and reporting, XML marshalling or un-marshalling, or mutually agreed to vertical industry extensions).

There are two types of extension allowed, **`preprocessor`** and **`postprocessor`**. If more than one included extension is defined of a given type they will be handled in the order that the extensions appear within the CAM template.

Further ideas for implementing extensions and example syntax are provided in the Addendum B of this document.

3.9.1 as:Extension

This is a hook to enable any extension to be included in the CAM Template. It may contain any valid XML from any defined source. Any number of extensions may be defined. Any process dependencies must be defined by the CAM processor supporting the Extension.

For Java implementations of CAM the Apache Maven linkage approach provides a default configuration method for associating external process handlers with the default CAM processor.

An example of an extension is provision of a lookup() function. This can be tailored to suit the particular domain and/or local application needs.

3.9.2 Preprocessor Extensions

Preprocessor extensions are run **after** the CAM template has been read in to the processor and after any pseudo-variables have been defined for the run. Any includes of any type are also completed before the extensions run. They are run **before** any BusinessUseContext rules are applied to the Structure in question.

In order to run the processor must supply an API to allow the preprocessor extension access to the complete CAM template and also to any input file that has been supplied to the processor. The preprocessor may then update either of these items before completion. A method to pass back any errors to the processor for onward communication must be provided.

3.9.3 Postprocessor Extensions

Postprocessor extensions are to be run after all the BusinessRulesContext rules have been completed. Processors are at liberty to provide an option as to whether extensions are run in the case of errors occurring during the core processing.

As with the preprocessors there are requirements to be able to access both the CAM template after any processing and the input file that has been processed. Each extension may change these and return them via the API for either the processor to complete work or to pass onto further extensions. A method to pass back any errors to the processor for onward communication must be provided.

3.9.4 as:include

The include provided outside the AssemblyStructure and BusinessUseContext elements is purely to allow as:Extension elements to be included.

In addition note that the as:include may optionally specify the ignoreRoot=”yes” attribute. This permits inclusion of XML fragments that are not well-formed, by allowing a dummy root element to be used to ensure the fragment is well-formed – but then the dummy root element is ignored.

e.g. :

```
<tempRoot>
   <not_well_formed_by_itself/>
   <tag1_include/>
   <tag2_include/>
   <well_formed>
     <tag3_include/>
   <well_formed>
</tempRoot>
```

So tempRoot will be ignored

3.9.5 Template Location defaulting

This provides the ability to associate from an XML instance to the CAM template that validates it. A URL is provided for the CAM template location. A CAM processor therefore can locate and validate XML directly. The syntax for this is:

asi:templateLocation="[URL]">

and the namespace declaration is:

xmlns:asi="http://www.oasis-open.org/committees/cam/instance"

and these should occur on the root element of the XML instance.

3.9.6 Selection of Assembly Structure

When a template contains more than one structure instance (such as different versions of the same structure) it is necessary to provide the ability to dynamically select which structure to apply to an XML instance for validation. One option is to pass in a CAM parameter. However this advanced feature permits the use of an xpath attribute onto the Structure element that then uniquely identifies the ID value of the relevant structure that should be used to validate the message (this can optionally be overridden by the structure ID name being passed in from outside the template). This first matching XPath expression that returns true is then selected for use.

The XML below provides an example. The xpath expression effectively equates to true if the XML instance contains the matching relevant structure item, and / or associated value.

```
<as:AssemblyStructure>
<as:Structure ID="ex_1" taxonomy="XML"
xpath="/ex:example">  <!-- Xpath check here -->
<ex:example>
<ex:test name="Fred">
</ex:example>
</as:Structure>

<as:Structure ID="new_1" taxonomy="XML"
xpath="/new:example">  <!-- Xpath check here -->
<new:example>
<new:test name="%Fred%">
<new:inside>%value%</new:inside>
</new:example>
</as:Structure>
</as:AssemblyStructure>
```

3.10 Future Feature Extensions

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

W3C RIF and OMG PRR Rule Support

The ability to add extensions to the base CAM templates means that common rule syntax approaches can be exploited easily to augment the base XML content validations that CAM provides. W3C Rule Interchange Format (RIF) and OMG Production Rule Representation (PRR) are both examples of such extended rules syntax that can be used to augment the basic built-in XPath support and CAM functions to add more complex logic handling. Examples of these techniques will be developed for future use.

RDF / OWL support

The ability to use RDF / OWL 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 to the OASIS CAM TC site 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.

XACML support

In many ways the CAM context mechanisms mirror the ability to include or exclude content as a filtering style operation between the input and output. An extension to support XACML (eXtensible Access Control Markup Language) syntax is there a natural addition to CAM processing. CAM functions can simplify the creation and coding of XACML while being able to call an XACML extension.

A. Addendum

A1.1 CAM schema (W3C XSD syntax)

This item is provided as a reference to the formal specification of the XML structure definition for CAM 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.   For specific details of the latest XSD documentation please reference 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://www.jcam.org.uk). See document download area from OASIS website: http://www.oasis-open.org/committees/cam

In addition OASIS may provide a static location to the reference CAM XSD schema under http://docs.oasis-open.org/cam once an approved specification is available.

A1.2 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.

A1.3 Processing Modes and Sequencing

Non-normative

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).

B. Addendum

B1.1 CAM extension mechanism example

This item illustrates the approach using Apache Maven linker technology to implement the component and Extension mechanism in CAM as implemented in the jCAM open source tool.   It also shows how alternative strict and lax XML conformance can be optionally configured via this mechanisms.

Figure B1.1.1

```
<container>
  <component-implementation class="uk.org.jcam.processor.dataObjects.Template" />
  <component-implementation class="uk.org.jcam.processor.dataObjects.DataFile" />
 <!-- <component-implementation
class='uk.org.jcam.processor.validator.DefaultValidator'/>
  -->
 <!-- <component-implementation
class='uk.org.jcam.processor.validator.UnOrderedValidatorLax'/>
  -->
  <component-implementation
class="uk.org.jcam.processor.validator.UnOrderedValidatorStrict" />
  <component-implementation class="uk.org.jcam.processor.trimmer.DefaultTrimmer"
/>
  <component-implementation class="uk.org.jcam.processor.adorner.DefaultAdorner"
/>
  <component-implementation
class="uk.org.jcam.processor.transformer.XSLTransformer" />
  <component-implementation class="uk.org.jcam.drools.DroolsDataValidator" />
  <component-implementation class="uk.org.jcam.groovy.GroovyDataValidator" />
  <component-implementation class="uk.org.jcam.beanshell.BeanShellDataValidator"
/>
</container>
```

C. Acknowledgements

The views and specification expressed in this document are those of the authors and are not necessarily those of their employers. The authors and their employers specifically disclaim responsibility for any problems arising from correct or incorrect implementation or use of this design.

The following individuals have participated in the creation of this specification and are gratefully acknowledged.

Participants:

Fred Carter	AmberPoint	CAM TC member
Chris Hipson	BTplc	CAM TC member
Martin Roberts	BTplc	CAM TC member
Hans Aanesen	Individual	CAM TC member
Ram Kumar	Individual	CAM TC member
Joe Lubenow	Individual	CAM TC member
Colin Wallis	New Zealand Government	Member, CIQ TC
David Webber	Individual	Chair CAM TC
Tom Rhodes	NIST	CAM TC member
Bernd Eckenfels	Seeburger, AG	CAM TC member
Paul Boos	US Dept of the Navy	CAM TC member

D. Non-Normative Text

Non-normative items are noted as such in the body of the specification as applicable. Possible Future Extensions are noted in that section above. Also a separate document is maintained by the CAM TC of experimental and extension items that are under consideration for inclusion in future versions of the specification. The latest public version of that draft non-normative items document is available from the committee area web site.

E. Revision History

[optional; should not be included in OASIS Standards]

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.
Revised Committee Draft		0.18	10th December 2004	DRRW	Revisions from comment period, corrections, and bug fixes to examples. Added Table of Figures index.
		0.19	4th January, 2005	DRRW	Layout changes to align with new OASIS document template formatting and logo. Update figure 4.1.2 to reflect latest schema, and also 4.5 for noun content referencing. Add addendum glossary of terms and abbreviations.
Revised Committee Draft	1.1	0.01	25th May, 2006	DRRW	New revised specification to reflect extensible model and architecture.
	1.1	0.02	27th June 2006	MR	Explicit corrections to line up with implementable features and also explicit definition of normative and non-normative sections.
	1.1	0.03	28th June 2006	MR	Included section on extensions (plug-ins).
	1.1	0.04	4th July 4, 2006	DRRW	Refined text, general edits.
	1.1	0.05	27th July, 2006	DRRW	Revise examples + figure captioning
	1.1	0.06	5th Sept 2006	MR	Issues around Order tackled

Status	Version	Revision	Date	Editor	Summary of Changes
	1.1	0.06	12th September 2006	DRRW	Changes consolidation and clean-up edits
	1.1	0.07	12th Sept 2006	MR	Schema Diagram Updated, Appendix re-factored, extensions approach re-worked
	1.1	0.08	15th Sept 2006	DRRW / MR	Edits and changes for accuracy. Import Function refined, Date comparison functions amended. W3C RIF and OMG PRR notes
	1.1	0.09	21st October 2006	MR / DRRW	Very Simple Extensions added. Align date masks with Java SDF. Correct examples XML
	1.1	0.10	24th October 2006	DRRW	Refine masks mechanism details, including both date and numeric masks.
	1.1	0.11	2nd November 2006	DRRW/ MR	Minor editing corrections and fixes to mask details, handling of quote characters and non-normative clarification of psuedo-variables.
	1.1	0.12	15th February 2007	DRRW	Revised to include comments from OASIS member 60 day review period (changes noted in comment review log document).
	1.1	0.13	5th March 2007	DRRW	Revised to use new OASIS document template and include TC member comments prior to formal Committee Specification ballot (changes noted in comment review log document).
	1.1	0.14	8th March 2007	DRRW	Cosmetic edits/fixes to URLs and layout to meet OASIS document specification, template and site requirements.