

Guidelines for Development of A Standard for Energy Transactions in XML (aseXML)

Version No: 3.3

FINAL

- 1. INTRODUCTION.....5**
 - 1.1 BACKGROUND5
 - 1.2 APPROACHES TO STANDARD DEVELOPMENT5
 - 1.3 DOCUMENT PURPOSE5
 - 1.4 TARGET AUDIENCE5
 - 1.5 REFERENCE DOCUMENTATION6
 - 1.6 FORMATTING CONVENTIONS.....6
 - 1.7 ASEXML CONCEPTUAL MODEL AND TERMINOLOGY6
 - 1.8 DOCUMENT STRUCTURE9
 - 1.9 REVISION HISTORY10
- 2. GENERAL.....15**
 - 2.1 DTDs VS SCHEMAS.....15
 - 2.2 USE OF SCHEMA VALIDATING PARSERS15
 - 2.3 ELEMENTS VS ATTRIBUTES15
 - 2.4 USE OF ENUMERATIONS.....16
 - 2.5 CODES VS DESCRIPTIONS17
 - 2.6 USE OF LINE TERMINATORS.....18
 - 2.7 THE SPIRIT OF ASEXML.....18
 - 2.8 CONTAINER ELEMENTS FOR REPEATED ELEMENTS19
 - 2.9 MAINTAINING ELEMENT ORDER.....19
- 3. VERSION CONTROL20**
 - 3.1 XML AND VERSIONING20
 - 3.1.1 *Options For Adding Version Information To XML*20
 - 3.1.2 *Namespaces*.....21
 - 3.1.3 *Namespace Granularity*.....21
 - 3.1.4 *XML Schemas*.....22
 - 3.2 ASEXML AND VERSIONING23
 - 3.2.1 *Guiding Principles*23
 - 3.2.2 *Role Of Versioning*24
 - 3.2.3 *Adding Version Information*25
 - 3.2.4 *Namespaces*26
 - 3.2.5 *Release Identifiers*26
 - 3.2.6 *Schemas*.....28
 - 3.2.7 *Version Attributes*.....28
 - 3.2.8 *Selecting type of versioning*.....29
 - 3.2.9 *Selecting Elements To Version*31
 - 3.2.10 *“Backwards compatible” changes*.....32
 - 3.3 USING DEVELOPMENT IDENTIFIERS33
 - 3.3.1 *Scenario*.....33
 - 3.3.2 *Sequence of Events*.....33
 - 3.4 ARCHITECTURE IMPLICATIONS OF ASEXML RELEASE MIGRATIONS.....36
 - 3.4.1 *Accepting aseXML Messages*36
 - 3.4.2 *Producing aseXML Messages*.....37
 - 3.4.3 *Minimising Code Branches*.....37
 - 3.4.4 *Release Example*.....38
 - 3.5 ASEXML VERSIONING STEP BY STEP39
 - 3.5.1 *The Initial Message*40
 - 3.5.2 *The Content Model Changes For An Isolated Element*.....41
 - 3.5.3 *The Content Model Changes For A Shared Element*.....42
 - 3.5.4 *The Content Model Changes For A Shared Element (2)*43

3.5.5	<i>The Structure Changes For A Shared Concrete Type</i>	44
3.5.6	<i>The Optionality Of An Element is Changed</i>	45
3.5.7	<i>Version Attribute Added</i>	46
4.	NAMESPACES	47
4.1	ASEXML NAMESPACE FORMAT	47
4.2	DEFAULT NAMESPACES	47
4.3	NAMESPACE PREFIXES	48
5.	SCHEMA ORGANISATION	49
5.1	SCHEMALOCATION URLs	49
5.2	TRANSACTION FILES	50
5.3	SCHEMA INCLUSION	51
5.4	COMMON SCHEMAS	51
5.5	ELEMENTS/TYPES	52
5.6	TRANSACTION ELEMENTS	52
5.7	ATTRIBUTES	53
6.	SCHEMA FEATURES	54
6.1	XML DECLARATION	54
6.2	ANONYMOUS vs NAMED TYPES AND DATA DICTIONARIES	54
6.3	ANNOTATIONS	54
6.4	SIMPLE TYPES	55
6.5	HANDLING FUEL SPECIFIC VARIATIONS	55
6.6	ASEXML ATTRIBUTES	55
6.6.1	<i>Default Values</i>	56
6.6.2	<i>ID And IDREF</i>	56
6.7	ELEMENT AND ATTRIBUTE QUALIFICATION	56
6.8	HANDLING EMPTY STRING ELEMENTS	56
6.8.1	<i>Nilable string element</i>	57
7.	INSTANCE DOCUMENTS	59
7.1	XML DECLARATION	59
7.2	DEFAULT NAMESPACES	59
7.3	SCHEMALOCATION ATTRIBUTE	59
7.4	DECLARING NAMESPACES FROM THE XML STANDARDS	59
8.	TRANSPORT, ENVELOPE OR TRANSACTION	60
8.1	TRANSPORT	60
8.2	ENVELOPE	62
8.3	TRANSACTION	62
9.	ENVELOPE	63
9.1	INTRODUCTION	63
9.2	<HEADER> SUB-ELEMENT	63
9.2.1	<i><From>, <To> (Mandatory)</i>	64
9.2.2	<i><MessageID> (Mandatory)</i>	64
9.2.3	<i><MessageDate> (Mandatory)</i>	64
9.2.4	<i><TransactionGroup> (Mandatory)</i>	64
9.2.5	<i><Priority> (Optional)</i>	64
9.2.6	<i><SecurityContext> (Optional)</i>	65
9.2.7	<i><Market> (Optional)</i>	65
9.3	<TRANSACTIONS> SUB-ELEMENT	65
9.3.1	<i>transactionID (Mandatory)</i>	65
9.3.2	<i>transactionDate (Mandatory)</i>	66

- 9.3.3 *initiatingTransactionID (Optional)* 66
- 9.4 FUTURE ENVELOPE MODIFICATIONS..... 66
- 9.5 A SAMPLE ASEXML MESSAGE 67
- 10. ACKNOWLEDGEMENT MODEL 68**
 - 10.1 INTRODUCTION 68
 - 10.2 TRANSACTION EXCHANGES VS TRANSACTION ACKNOWLEDGEMENTS .. 68
 - 10.3 MESSAGE ACKNOWLEDGEMENT 68
 - 10.3.1 *initatingMessageID (Mandatory)*..... 69
 - 10.3.2 *receiptID (Optional)* 69
 - 10.3.3 *receiptDate (Mandatory)*..... 69
 - 10.3.4 *status (Mandatory)* 69
 - 10.3.5 *duplicate (Optional)* 70
 - 10.4 TRANSACTION ACKNOWLEDGEMENT 70
 - 10.4.1 *initatingTransactionID (Mandatory)*..... 70
 - 10.4.2 *receiptID (Optional)* 70
 - 10.4.3 *receiptDate (Mandatory)*..... 71
 - 10.4.4 *status (Mandatory)* 71
 - 10.4.5 *duplicate (Optional)* 71
 - 10.4.6 *acceptedCount (Optional)* 72
 - 10.5 EXCHANGING ACKNOWLEDGEMENTS..... 72
 - 10.6 HANDLING DUPLICATES 72
 - 10.7 A SAMPLE ASEXML TRANSACTION EXCHANGE 74
- 11. ERROR REPORTING AND THE <EVENT> ELEMENT 75**
 - 11.1 CLASS ATTRIBUTE (OPTIONAL) 75
 - 11.2 SEVERITY ATTRIBUTE (OPTIONAL) 76
 - 11.3 <CODE> SUB-ELEMENT (MANDATORY) 76
 - 11.4 <KEYINFO> SUB-ELEMENT (OPTIONAL)..... 77
 - 11.5 <CONTEXT> SUB-ELEMENT (OPTIONAL)..... 77
 - 11.6 <EXPLANATION> SUB-ELEMENT (OPTIONAL) 78
 - 11.7 <SUPPORTEDVERSIONS> SUB-ELEMENT (OPTIONAL) 78
 - 11.8 RESERVED EVENT CODES 79
- 12. GENERIC TRANSACTION EXCHANGES 82**
 - 12.1 TABLE REPLICATION..... 82
 - 12.2 REPORTS 84
- 13. SUPPORT FOR CSV FORMAT DATA..... 85**
 - 13.1 FORMAT CONSTRAINTS 85
 - 13.2 LINE TERMINATOR 85
- 14. ACCESSING ASEXML SCHEMAS AND INSTANCE DOCUMENT EXAMPLES..... 86**
- 15. MESSAGE SERVICES..... 86**
 - 15.1 ASEXML FTP HOKEY-POKEY PROTOCOL 86
 - 15.2 USING ASEXML WITH EBXML MESSAGING SERVICE 86
 - 15.3 ASEXML BINDING WITH SMTP 86

1. INTRODUCTION

1.1 BACKGROUND

The combined Gas and Electricity IT Architecture Working Group of Australia has adopted a number of recommendations in the area of business-to-business electronic data interchange (see document references in section 1.5). The thrust of this work is an acceptance of XML to describe business transactions and the Internet to exchange them.

The working group has commissioned the development of this document in order to further the standardisation of the transactions required within the Australian energy market.

1.2 APPROACHES TO STANDARD DEVELOPMENT

There are various approaches that may be adopted in the development of a standard.

Centralised approaches typically involve the formulation of a representative committee that drafts the specification, followed by its implementation by the participants. The aim is for up-front consensus to avoid future interoperability problems, but the process often suffers from bureaucratic delays.

De-centralised approaches allow individuals to develop working prototypes and have these ratified by an authorising committee. A high degree of parallelism may be achieved, but broader acceptance is contingent on the standard meeting the requirements of all involved.

Given the tight timeframes established for full retail contestability, a de-centralised model to standards development has been adopted. The aim is to harness the collective intellectual property of the industry with individuals focussing on those areas where they perceive the most benefit.

1.3 DOCUMENT PURPOSE

The purpose of this document is thus to establish sufficient infrastructure to allow the independent development of portions of the specification and their combination in an efficient manner. Given the process used, this document will of necessity evolve over time and should be considered a “work in progress”.

1.4 TARGET AUDIENCE

This document is designed for technical and software development staff responsible for systems implementing the aseXML standard.

It is assumed that readers of this document are familiar with the standards below.

1. Extensible Markup Language (XML) 1.0 (www.w3.org/TR/REC-xml)

2. Namespaces in XML (www.w3.org/TR/REC-xml-names)
3. XML Schema Part 1: Structures (www.w3.org/TR/xmlschema-1)
4. XML Schema Part 2: Datatypes (www.w3.org/TR/xmlschema-2)
5. XSL Transformations (XSLT) Version 1.0 (www.w3.org/TR/xslt)

1.5 REFERENCE DOCUMENTATION

The following documents may be of use for background information.

1. Combined Gas & Electricity IT Working Group White Papers (<http://www.asexml.com/>)
2. XML Schemas: Best Practices (<http://www.xfront.com/BestPracticesHomepage.html>)
3. ISO/IEC 11578:1996 – “Information technology – Open Systems Interconnection – Remote Procedure Call”

1.6 FORMATTING CONVENTIONS

This paragraph demonstrates the appearance within this document of any text defining a requirement for conformance to aseXML.

Any text representing the literal value used for elements or attributes will be shown in fixed pitch font, e.g. `<TransactionGroup>`.

1.7 aseXML CONCEPTUAL MODEL AND TERMINOLOGY

Words such as “transaction”, “message”, “acknowledgement” and “gateway” are commonly used in a wide variety of contexts within the Information Technology Industry.

It is thus important to understand their use within aseXML. Figure 1 below presents the conceptual model used by aseXML and its use of such terms. The terms appearing on the diagram are defined in subsequent paragraphs. They will be further expanded in subsequent chapters and sections of this document.

A **transaction** is a one-way exchange of information between applications within communicating end systems.

A **transaction exchange** is the exchange of one or more transactions between applications. It consists of a **request transaction**, followed by zero or more **response transactions**. Typically transaction exchanges follow a request/single response model.

For each transaction of a transaction exchange, the receiving application responds with a **transaction acknowledgement**. A transaction acknowledgement allows tracking of the transaction’s progress and flags the receiver’s commitment to process it. It may also be used to carry error information with regards to the corresponding transaction.

also be used to carry error information with regards to the corresponding message.

In order to prevent circular acknowledgements, any message containing a message acknowledgement is not itself acknowledged.

An **aseXML Gateway** is responsible for validating aseXML messages and routing them to external systems, or the contained transactions to the appropriate internal application. In order to exchange messages with an external system, the gateway uses the facilities offered by one or more **transport** layers.

A transport layer is assumed to provide reliable delivery of payloads. aseXML acknowledgements should thus be considered in the context of message or transaction auditing and tracking rather than as part of a reliable delivery mechanism.

An example of the use of the above terminology is given below, with this example used as the basis for other examples in this document.

Application – NMI Data Access

Transaction Group - NMID

Transaction Exchanges – NMI Discovery, NMI Standing Data

Transactions –

NMI Discovery : NMI Discovery Request, NMI Discovery Response

NMI Standing Data : NMI Standing Data Request, NMI Standing Data Response

1.8 DOCUMENT STRUCTURE

Chapter	Area Covered
2	General requirements needed prior to a detailed discussion of XML Schema organisation
3	Version control within aseXML. It is necessary to define the versioning mechanism to be used as it impacts on naming standards
4	Namespace use within aseXML
5	Source file management and element naming for aseXML Schemas
6	Use of XML Schema features within aseXML
7	Format requirements for instances of aseXML documents
8	Distinction between XML defining transactions and XML needed to carry information about the process and its transactions
9	XML Envelope to be used within aseXML
10	Transaction Exchange Model for aseXML, including acknowledgement mechanisms
11	Error and Event Handling
12	Generic Transaction Exchanges
13	Support for CSV format data
14	How to obtain schemas and examples for aseXML
15	References to documentation of messaging services to exchange aseXML documents

1.9 REVISION HISTORY

Version	Date	Who	Comments
0.1	25/09/2000	Michael Leditschke	Initial draft
0.2	02/10/2000	Michael Leditschke	Rework with single namespace
0.3	04/10/2000	Michael Leditschke	Simplify version identifiers Add special text format for requirements
0.4	09/10/2000	Michael Leditschke	Add additional element naming guidelines
0.5	31/10/2000	Michael Leditschke	Final review before release to IT WG Note: Diagrams are still to be completed.
0.6	09/10/2000	Michael Leditschke	Add diagrams Revised text of chapter 8
0.7	19/12/2000	Michael Leditschke	Schemas now based on 24 th October 2000 candidate recommendation Clarify the use of the “ref” construct for global elements Remove restriction on the encoding scheme used. All implementations must support UTF-8 to comply with the Extensible Markup Language (XML) 1.0 specification, and ASCII is a subset of UTF-8. Sample schemas and instance documents no longer contained in this document. Reference to the appropriate URLs is provided Added caveat to codes vs. descriptions allowing no description where code/description mappings known to businesses Added chapter 10 on the aseXML Acknowledgement Model

Guidelines for Development of A Standard for Energy Transactions in XML (aseXML)

Version	Date	Who	Comments
	06/03/2001		<p>Updated chapter 9 on the aseXML Envelope to reflect envelope used for MSATS – remove use of the term “Interim” in the header</p> <p>Added section 1.7 on transaction terminology in Introduction</p> <p>Expanded section 5.4 on common schemas</p> <p>Added section 6.2 on use of anonymous types</p> <p>Changed document title to avoid Standards Australia trademarks</p> <p>Expanded section 6.3 on use of annotations in line with desire to automatically generate data dictionaries from the schemas. Removed chapter on documentation.</p> <p>Added chapter 11 to more fully cover error reporting</p>
0.8	16/03/2001	Michael Leditschke	<p>Allow message and transaction level acknowledgements in a single message</p> <p>Namespace usage within schemas now consistent with reference 2.</p>
0.9	20/03/2001	Michael Leditschke	<p>Rename <Location> element of <Event> to <KeyInfo> and change description</p> <p>Add text indicating what severity levels should accompany acknowledgements</p>
1.0	23/03/2001	Michael Leditschke	Reformat as FINAL
1.1	30/05/2001	Michael Leditschke	<p>Minor editorial changes</p> <p>Acknowledgements now use the term <code>receipt</code> rather than <code>request</code></p> <p>Chapter 13 now refers to the URL that provides the entry point to information with regard to aseXML</p> <p>Add recommendation with regard to the use of UUIDs for <code>messageIDs</code>,</p>

Guidelines for Development of A Standard for Energy Transactions in XML (aseXML)

Version	Date	Who	Comments
			<p>transactionIDs and receiptIDs</p> <p>Add reference to ISO 11578.</p> <p>receiptID attribute on acknowledgements is optional in case where message or transaction is rejected</p> <p>Add comment with regard to generation of a new messageID or transactionID in the case of a rejection</p> <p>Add <Market> element to the message header to allow identification of the energy market in which the transactions should be considered</p> <p><Event> attributes are now optional with default values.</p> <p>Rearrange standard event codes such that they are unique. Add a few additional standard codes.</p> <p>Change <Event> class attribute value of "Data" to "Application" to more closely match its intended purpose</p> <p>Expand section on aseXML terminology and include information on the aseXML conceptual model. Remove the term "business process" and replace with "application" or "transaction group" to be consistent with the model.</p> <p>Add section 10.2 to clarify the role of transaction acknowledgements in transaction exchanges.</p> <p>Added additional text to section 10.5 to further clarify the issues associated with exchanging acknowledgments.</p> <p>Schemas should now use the 02/05/2001 XML Schema recommendation.</p> <p>The requirement for all attributes to be mandatory has been removed.</p>

Guidelines for Development of A Standard for Energy Transactions in XML (aseXML)

Version	Date	Who	Comments
			<p>Added <code>duplicate</code> attribute definition and description to acknowledgements</p> <p>Added <code>acceptedCount</code> attribute definition and description to transaction acknowledgement</p>
1.2	27/06/01	Michael Leditschke	<p>Introduced the concept of generic transaction exchanges that can appear within multiple <code>TransactionGroups</code>, e.g. reports and table replication</p> <p>Relaxed the restriction in section 1.7 that a transaction exchange may only appear in one <code>TransactionGroup</code> to allow for generic transactions</p> <p>Added chapter 12 on generic transaction exchanges</p> <p>Added additional standard event codes in section 11.8</p> <p>Completed hanging sentence in section 10.2 (thanks James)</p>
1.3	22/08/01	Michael Leditschke	<p>Added a comment clarifying the need for uniqueness with <code>MessageIDs</code> and <code>transactionIDs</code> (sections 9.2.2 and 9.3.1)</p> <p>An event severity of "Information" should be used, in the absence of any other circumstances, with a code value of 0 (section 11.2)</p> <p>Added section 13.2 to document the line terminator to be used with CSV data</p> <p>Added section 2.7 on the desire for a single transaction set per business process</p>
2.0	13/05/02	Michael Leditschke	<p>Minor editorial corrections</p> <p>Correct example in section 10.7</p> <p>Incorporate change proposals</p> <ol style="list-style-type: none"> 1. Schema file naming 2. Event code ranges

Guidelines for Development of A Standard for Energy Transactions in XML (aseXML)

Version	Date	Who	Comments
			<p>3. "The Spirit of aseXML"</p> <p>4. Event code 999</p> <p>5. Repeating elements</p> <p>6. Maintaining element order</p> <p>NB. An additional sentence has been added to the text of proposal 6 to include reference to the term "parallel design", which is sometimes used for this design pattern.</p> <p>7. Enumerations</p> <p>8. Enhanced versioning</p>
2.1	14/10/02	Bibhakar Saran	<p>Incorporated change proposals:</p> <p>2.1. Version attribute for derived types</p> <p>2.2. Add aseXML binding details for ebXML messaging and other relevant protocols</p> <p>2.3. Add error code 206</p>
3.0	6/10/03	Darren Field	<p>Added information on patch releases (section 3.2.5).</p> <p>Clarified handling of duplicate messages and transactions (section 10).</p>
3.1			Rejected
3.2	16/11/04	Andrew Screen	<p>Clarification of when version attributes are updated (section 3.2.9)</p> <p>Updates to enumerations handling (section 2.4)</p>
3.3	23/05/2007	Nada Reinprecht / Andrew Screen	<p>Handling of empty strings (Section 6.8)</p> <p>Update for the handling of default and optional versioning (section 3.2.7 & 3.2.8 [new] 3.2.9 [old 3.2.8]).</p>

2. GENERAL

2.1 DTDs VS SCHEMAS

The data dictionary and transactions will be expressed in the language of XML schemas rather than DTDs.

This follows the trend towards the use of schemas in much of the work currently being undertaken on the Internet.

Schemas will use the 2nd May 2001 XML Schema recommendation until such time as any new version of the specification reaches recommendation status.

2.2 USE OF SCHEMA VALIDATING PARSERS

A schema validating parser will process incoming XML documents in order to ensure full compliance to the aseXML standard.

This parsing should occur as early as possible, preferably prior to application processing, in order to ensure the timeliest rejection of invalid transactions.

Use of such a parser may also remove some of the validation burden from the receiving application and assist in ensuring consistent industry wide validation.

2.3 ELEMENTS VS ATTRIBUTES

There have been many debates within the XML community with regard to the representation of data items in elements as opposed to attributes. Many XML standards such as XSL provide equivalent functionality for both and often the choice is a matter of philosophical preference.

The main differences between attributes and elements in this context are that

- Attributes can only be of simple types, whereas elements may be of complex types.

Complex data items such as addresses are thus not appropriate candidates for attributes.

- Versioning of attributes is difficult to achieve

By its nature, it is difficult to attach versioning to an attribute, whereas an element can easily carry a version attribute. In addition, mechanisms such as the `<choice>` tag in schemas are only available for elements and not for attributes.

Approaches to deciding what information belongs where cover a broad range including the following:

- Use elements for content and attributes for metadata about the content.

An example might be to use an element for a bid structure and an attribute of this element for the bid date.

- Use attributes where there is no likelihood of further data refinement otherwise use elements.
- Where there is no other deciding factor, use an attribute rather than an element because of its more concise syntax.

Whilst it is recognised that no particular approach is more “correct” than any other, one approach needs to be selected to provide consistency across the transactions within aseXML. The rules below will thus be used to determine when to use elements and attributes.

- **Use elements for content and attributes for metadata about the content.**
- **If there is any chance of further data refinement, use an element.**
- **If there is the possibility that multiple versions may need to co-exist, use an element.**
- **If in doubt, use an element.**

2.4 USE OF ENUMERATIONS

One feature of XML Schemas, called an enumeration, limits the contents of an element or attribute to a finite set of values. Use of enumerations in aseXML schemas is desirable to provide global documentation of this set of values in an enforceable manner.

It is recognised, however, that where the possible set of values is changing frequently, enumerations may cause problems in areas such as versioning. In addition, determining the valid set of values may more readily be handled in application code, particularly where processing logic depends on the value. The disadvantage of application-based validation is that it must be implemented by all participants rather than once in the schema.

Schema designers are thus encouraged to use enumerations provided the values are stable. As a general rule of thumb, if the set of valid values changes as a result of an administrative function, an enumeration should NOT be used, for example registration of a new participant. If the set of valid values changes as a result of industry-wide consultation, however, enumerations may be considered, for example addition of new tranches.

Special Case

Whilst the above is in general preferable, enumerated lists are in use within the schema where it is possible for the values to change between schema releases. In order to handle this, a pragmatic solution has been to put the enumerations into a separate file called enumerations.xsd and remove the release identifier, so that it can be updated based upon a coordinated business process, separate from the schema release. The risks associated with this solution are deemed to be less than the problems caused by not updating the list.

Where there are only two possible values for an enumeration, the in-built Boolean type, NOT an enumeration, should be used. In this case, the element name should carry the meaning of the “true” value. An example is shown below of a row status data field that can have the content values of “Active” or “Inactive”.

USE

```
<RowActive>true</RowActive>
```

NOT

```
<RowStatus>Active</RowStatus>
```

Descriptive terms rather than abbreviations should, in general, be used for enumeration values. The motivation for this is to achieve readability of the resulting XML, recognising that mapping of the enumeration values to internal values is likely by both the sender and receiver. Note that this requirement does not rule out the use of industry accepted code sets, such as those used to specify Australian addresses.

An enumeration should only have one value per logical meaning. For instance, if an enumeration had a value of “Energised”, it would not be acceptable to also include a value of “En” or “E” to represent abbreviated forms of “Energised”. Similarly, a value of “Powered”, if it implied the same logical meaning as “Energised”, would not be acceptable.

2.5 CODES VS DESCRIPTIONS

Where codes or alphanumeric identifiers have an equivalent textual value, it is desirable that both the mnemonic and its equivalent description be carried by a transaction.

This will enhance human readability of the transaction as well as information display and validation. This approach is particularly important where codes are specific to a particular participant.

Where mechanisms are in place for the exchange between businesses of the code/description mapping information, use of descriptions within transactions should be considered optional.

When included, a description will be carried either as a separate sub-element or as an attribute of the element. By preference, the sub-element <Description> or the attribute “description” should be used.

An example is given below.

```
<DistributionLossFactor>  
    <Code>QLD23</Code>  
    <Description>Brisbane Metro</Description>  
</DistributionLossFactor>
```

or

```
<DistributionLossFactor code="QLD23"  
    description="Brisbane Metro"/>
```

In line with section 2.4, enumeration of the possible values for codes and equivalent descriptions should be included in the schema where appropriate.

2.6 USE OF LINE TERMINATORS

Schemas and instance documents should incorporate line terminators to assist in human readability, subject to issues related to data volume.

The start and end tags of elements containing sub-elements should stand alone on a line, whilst the tags of elements not containing sub-elements may reside on a single line.

2.7 THE SPIRIT OF aseXML

This document focuses largely on the common infrastructure needed to allow the exchange of transactions (see section 1.7) However, it is equally important to realise that in developing aseXML, there is a strong desire that there should only be one set of transactions used for a given business process. The transactions thus need to be designed, or need to be modified over time, to accommodate variations between markets and fuel types.

The driver for commonality of transactions across different fuel types and markets is to minimise the requirement for different systems and business processes to be built both at the central hub and at the participant end. This desire to minimise cost in handling transactions between businesses is fundamental to the development of the standard.

Any party wishing to introduce new transactions to aseXML needs to ensure that there is not already an existing set of transactions that broadly covers the business process being addressed.

Conversely, care should be taken in using existing transactions to enable similar but subtly different business processes that sufficient documentation exists of these differences.

This desire, not only for a single transaction infrastructure, but a single set of transactions, has come to be referred to as the “spirit” of aseXML.

Compliance to the “spirit” of aseXML is an important aspect when considering whether proposed schema changes comply with the aseXML guidelines.

2.8 CONTAINER ELEMENTS FOR REPEATED ELEMENTS

Where an element carries a `minOccurs` attribute with a value greater than one, the repeated elements should be immediately enclosed by a container element, whose name reflects the nature of the grouping.

An example is shown below.

```
<FaultDescriptionComments>
    <Line>First line of comment</Line>
    <Line>Second line of comment</Line>
</FaultDescriptionComments>
```

The container element name will typically, though not necessarily, be plural as per section 5.5.

2.9 MAINTAINING ELEMENT ORDER

It is often the case that a given set of elements will appear in multiple places within aseXML, though with differing optionality. Typical examples of this might be report parameter formats for a particular transaction group or table replication formats with common elements.

In this situation, it is recommended that preference be given to maintaining the order of the elements across the multiple situations where they occur. Such an approach is sometimes referred to as “parallel design”. This should be contrasted against alternate options, such as ordering according to whether the elements are mandatory or optional.

Adoption of this recommendation is likely to assist in simplifying the design of applications designed to produce the formats.

3. VERSION CONTROL

3.1 XML AND VERSIONING

Ask ten XML practitioners how to handle the issues associated with XML versioning and you will undoubtedly receive ten divergent answers. Versioning is complicated by issues such available XML tools and programming techniques, version change rate, application development lifecycles and size of user base.

There are however some basic building blocks from which XML versioning schemes are generally constructed. This section provides some detail of these blocks. For those familiar with XML and XML Schemas, section 3.2 describes the specific way versioning is implemented in aseXML.

3.1.1 Options For Adding Version Information To XML

There are a number of ways to associate version information with XML. Each is discussed below. It should be noted they are not mutually exclusive and are often combined in practice.

- Maintain version information externally

Some transaction frameworks use bi-lateral agreements to document version requirements for exchanged documents. The documents themselves need not carry version information, or carry minimal information to confirm conformance to the agreement.

- Incorporate version information into element/attribute names

This approach has the advantage that different versions of the same element/attribute may co-exist in one schema, but requires micro-parsing of names to extract version information. Its effect is also marked in terms of application code, since it incorporates version information into the structure of the XML via its effect on element/attribute names.

- Attach version attributes to elements

This approach is commonly used, since version information can be viewed as metadata about the element. It is also less intrusive than the previous option, since the version information is in the content of the XML. Many of the XML recommendations employ version attributes. This approach does not lend itself to versioning of attributes.

- Incorporate version information into namespaces

By associating new versions of elements/attributes with different namespaces, namespace aware processing code can make the necessary logic adjustments for different versions. More detail on namespaces is provided in following sections.

3.1.2 Namespaces

Namespaces are an important concept when considering XML and versioning. Quoting from the “Namespaces in XML” specification,

“Software modules need to be able to recognise the tags and attributes which they are designed to process, even in the face of “collisions” occurring when markup is intended for some other software package using the same element type or attribute name.”

“An XML namespace is a collection of names, identified by a URI reference, [RFC2396], which are used in XML documents as element types and attribute names”.

Some XML standards such as [Scalar Vector Graphics](#) (Appendix F.3) and [Signature Syntax and Processing](#) have specified the use of multiple namespaces to detect different versions of the specification.

Others, such as [XSL Transformations](#) attach a version attribute to top level elements and define behaviour necessary to process XML documents that use different versions and mechanisms to add extensions to the base standard. In this manner, they avoid the need to change the namespace used.

The quotes above could be interpreted to mean that different versions of an element belong to different namespaces. Others argue for the use of namespaces in a broader sense, for instance a namespace for everything within aseXML regardless of version.

The jury is thus out as to what the XML community think is the best way to incorporate namespaces in a versioning strategy, if at all.

3.1.3 Namespace Granularity

Assuming namespaces are to be used as part of a versioning strategy, one of the design decisions to be made is how many namespaces to use. The following table summarises the options and their advantages and disadvantages.

Approach	Granularity	Advantages	Disadvantages
Single namespace	Coarse	Simple No need to use namespace prefixes in instance documents via use of default namespace	No granularity Alternate methods to track version variations within a document need to be considered
Namespace per element/attribute	High	Fine version control	<ul style="list-style-type: none"> • Large number of namespaces • Complex management at application level • Use of multiple namespaces complicates schema design and instance documents
Namespace per group of elements/attributes . transaction group	Medium	<ul style="list-style-type: none"> • Parallels likely participant support of portions of the specification • Reasonable granularity 	<ul style="list-style-type: none"> • Complex management at application level • Use of multiple namespaces complicates schema design and instance documents

The issue is largely a trade-off between simplicity and insulation from unnecessary change. Elements/attributes in one namespace are insulated to some extent from changes in other namespaces, but the penalty incurred is the need to manage versions of multiple namespaces.

3.1.4 XML Schemas

The “XML Schema” specification builds on the “Namespaces in XML” specification by providing a mechanism to define the elements and attributes belonging to a particular namespace. The particular namespace is referred to as the “target namespace”. To validate an element/attribute, a schema is needed whose target namespace matches the namespace of the element/attribute.

Thus, the question naturally arises “Given an element of a particular namespace, how do I obtain the corresponding schema?” Much of the debate has centred on the use of a URI to identify a namespace. Because one form of a URI is a URL, one approach is to use a URL for a namespace and provide the corresponding schema via the URL. Many

have argued against this, indeed the “Namespaces in XML” specification includes the sentence

“It is not a goal that it (a namespace name) be directly useable for retrieval of a schema...”

The designers of the schema specification did provide a partial answer to the question by defining a “schemaLocation” attribute that can be added to an element as a way for its instance document originator to provide assistance as to what the intended schema should be. The value of this attribute may be one or more namespace/URI pairs. It is the usual convention for the URIs to take the form of URL’s by which the schema for the namespace may be retrieved.

The schemaLocation attribute is optional and even if present may be ignored. Indeed, the specification goes on in “XML Schema Part 1: Structures (Section 4.3.2)” to allow schema processors to pick and choose from a variety of ways to retrieve schemas based on either the namespace or the schemaLocation, from either a local cache or the Internet.

3.2 aseXML AND VERSIONING

3.2.1 Guiding Principles

In selecting a versioning approach, aseXML has attempted to pick the “middle road” that ensures possible changes in versioning strategy are not precluded, while not unduly complicating the generation and processing of instance documents. There is some overlap in the techniques used, which will most likely disappear over time as a result of experience, version support in transport frameworks, and new standards addressing the issue of versioning XML.

The principles below have been used to guide the formulation of the approach.

1. Minimise the amount of version information within instance documents.

This ensures instance documents are simple to generate and read.

2. Add version information in a way such that it can be removed/ignored in the future.

This allows a smooth migration to standardised versioning techniques in the future without, where possible, invalidating existing instance documents.

3. Accommodate the need for applications to make processing decisions on the basis of version.

As discussed in section 3.2.2, any version mechanism must provide version information to applications. This should be done in a manner that is simple to handle programmatically.

3.2.2 Role Of Versioning

It is a subtle but important point to realise that the role of versioning in aseXML documents is twofold.

1. Application Logic Control

On one hand, application code must be aware of any variations in the structure or content of the XML elements with which it is dealing. Thus the first role of versioning within aseXML is to allow application code to make such processing decisions.

The important point to note is that the code is only interested in changes specific to its XML elements. If the effects of change are to be localised, changes elsewhere should have no impact on the ability of the application code to process or generate unchanged XML elements. Put another way, version information for a given XML element should only change when its structure or content does.

2. Instance Document Validation

On the other hand, in line with section 2.2, a validating parser must be able to check an incoming aseXML document against the relevant XML Schema. Because the document may contain multiple versioned elements, each of which having a different version history, the Schema must be capable of handling this.

Thus the second role of versioning within aseXML is to support the process of document validation.

- **Implications for XML Schemas**

aseXML uses an XML Schema to codify a cross-section of version information, with associated structure and content definitions, that was current at a given point in time. Each Schema is effectively a “snapshot” of the latest definitions at a point in time.

There are a few important implications of this approach.

1. The definition of an unchanged XML element will be included but unchanged across multiple snapshots.

Such an element may be delivered with the same version information under any of the snapshots in which it was captured.

2. Snapshot information need not be passed to application code.

Application code is not interested in snapshots per se. All the code requires is the XML fragment to be processed and the associated information indicating the specific version of the element with which it is dealing.

3. A snapshot contains only the latest definitions.

A given version of an element will only appear in a snapshot if it was the current version at the point in time of the snapshot. Put another way, it is not possible to deliver in a single document a combination of element versions that does not reflect historical reality.

- Implications for instance documents

In terms of instance documents, validation support should allow an incoming document to flag with which snapshot the sender believes it is compliant.

The receiver can use this information

1. to determine whether it supports the snapshot
2. to choose the XML Schema to be used by the parser to confirm compliance to the snapshot.

3.2.3 Adding Version Information

Each of the options discussed in section 3.1.1 is considered below in the light of sections 3.2.1 and 3.2.2.

- Maintain version information externally

Given the number of participants and the overheads of this process, this option was not considered appropriate for aseXML, especially in the initial stages of market development, where a high rate of change was envisaged. By not explicitly providing version information in the instance, this approach was also seen as adding complexity to the process of indicating version information to application code (see 3.2.1, point 3).

- Incorporate version information into element names

This approach was rejected because it did not allow version information to be easily removed or ignored and was considered to increase the complexity of producing instance documents (see section 3.2.1, points 1 and 2).

- Attach version attributes to elements

Version attributes are used by aseXML to fulfil the first role of versioning within aseXML (see section 3.2.2), which is to provide element version information to applications.

They may easily be ignored if necessary in the future and may easily be accessed by application code (see section 3.2.1, points 2 and 3). In addition, in the case where a fragment of an incoming document is forwarded to an application, version attributes easily travel with their corresponding element.

No versioning of attributes is supported independent of versioning on elements.

Section 3.2.7 provides further details of the use of version attributes.

- Incorporate version information into namespaces

Namespaces are used to fulfil the second role of versioning in aseXML (see section 3.2.2), which is to support the process of document validation.

Section 3.2.4 provides further details of the use of namespaces.

3.2.4 Namespaces

aseXML will use a single namespace to cover all elements within it, but will incorporate version information in the namespace, effectively using a new namespace each time the specification is changed.

Each namespace thus represents a snapshot of version information, as discussed in section 3.2.2.

Instance documents will qualify their top-level element with the aseXML namespace corresponding to a snapshot to which the document conforms.

It should be noted that, ignoring the aseXML namespace declaration, an instance document may be valid for multiple snapshots, and in this case, it is possible for it be delivered under any of them. Procedural rules may however constrain the allowable set of snapshots.

The reasons below were used in determining the use of namespaces by aseXML.

- Use of one namespace is in line with the section 3.2.1 point 1, that is simplicity of schemas and simplicity of instance documents.
- Some schema parsers (see section 3.1.4) may indirectly use namespaces as a way of locating the corresponding schemas, and hence information may be needed in the namespace to differentiate between versions.
- The version information may easily be frozen should the need disappear for its presence in the namespace.
- Given the large number of participants, and the varying timing of their IT development cycles, use of multiple namespaces was seen as adding an unnecessary layer of dependencies to the challenge of progressing version changes to the aseXML standard.

3.2.5 Release Identifiers

A release identifier is used to identify each version snapshot of aseXML. A release identifier starts with a lowercase “r” and is followed by a whole number, referred to as the release number.

Such an identifier is referred to as a production release, an example of which is given below.

r100

A released version of the schema may require a patch if an error is discovered, or update is required, following the schema release. A patch release will normally follow an abridged change process to allow the patch to be published quickly and outside the schema release schedule.

In order to identify a patch release, a patch extension will be appended to the affected production release, being separated from it by an underscore character. Such an identifier will be referred to as a patch release.

The first letter of the patch extension will be 'p', followed by a sequence number to identify the specific patch.

An example of a patch release is given below.

r100_p1

In order to develop new production releases, a development extension may be appended to the affected production release, being separated from it by an underscore character. Such an identifier will be referred to as a development release.

The first letter of the development extension will indicate the particular thread of development. It will be followed by a sequence number to allow identification of the stage of development within the thread.

An example of a development release is given below.

r100_a5

Use of development releases is highlighted in the section 3.3.

Whenever they appear, release identifiers will be separated from other text by an underscore character.

Release identifiers are incorporated into aseXML namespaces (see section 4.1) and provide the content of version attributes (see section 3.2.7).

Common language usage often sees reference to the "version of aseXML" or the "aseXML namespace", where technically a release of aseXML is intended. So, for instance,

"version r7 of aseXML"

is equivalent to

"the r7 namespace of aseXML"

which is equivalent to

"release r7 of aseXML"

or, for this release identifier,

“production release r7 of aseXML”.

All are synonyms for a particular snapshot of version information to be used for document validation.

3.2.6 Schemas

For each aseXML namespace, a corresponding XML Schema will be created.

Given that different products may use different strategies to obtain schemas, it is not possible to be prescriptive in this standard as to how the mapping between schema and namespace will be determined. In order to facilitate different approaches, however, the rules below will be used.

The URIs used in schemaLocation attributes will be URLs by which the schema may be obtained.

Given knowledge of the base portion of a schemaLocation URL, it will be possible to automatically generate the schemaLocation attribute corresponding to a namespace.

The root element of each instance document will provide a schemaLocation attribute for its corresponding aseXML namespace.

At first glance it may seem that dynamic fetching of schemas will not occur, since application changes must precede presentation of associated transactions for any meaningful work to be done. However, as discussed in section 3.3, a participant might receive a transaction for a version of aseXML not yet supported within their systems. In this case, there is still an obligation to parse the transaction as per section 2.2, in order to formulate an appropriate response.

3.2.7 Version Attributes

Version attributes will be attached to major aseXML elements. These may be used to provide application code with version information concerning the structure and content of these elements.

Version attributes will use the name `version` and contain a release identifier.

Each time changes are required within aseXML, a new release identifier will be created and assigned to those versioned elements affected by the changes.

A by-product of this process will be a new version snapshot, identified by the new release identifier, belonging to a new namespace, and whose structure and content are defined by the corresponding schema.

Multiple changes to a versioned element may be reflected in a single change in the release identifier carried in the element's `version` attribute.

Over time, each versioned element will be assigned a subset of the total set of release identifiers, based on its change history. Each release in this subset is referred to as a “release point” and indicates the release at which the contents of the element, and hence the associated application semantics, changed.

Indications of changes to non-versioned elements “bubble up” to the nearest enclosing versioned element. The scope of a change is thus limited to the most tightly enclosing versioned element.

This approach has several features.

- Application code for a given versioned element need only know how to generate and process a limited set of production releases, in line with section 3.2.1, point 3. Code should be structured to use `version` attributes to control variations in processing.
- Code written to generate a particular release of an element will not be invalidated when a new snapshot is created as a result of modifications in some other element.
- The presence of the `version` attribute allows future definition of how applications might process versions later than those supported, perhaps via a mechanism similar to “Forwards Compatible Processing” in the XSL Transformation specification (see section 3.1.2).
- Given that a query mechanism is available, an application having a given version of a component rejected (presumably because of lack of support within the recipient) may determine what versions are common between the two participants and use the highest version available.
- Examination of the schema for any snapshot will quickly reveal the release point of each versioned element, since the definition of the element will be carried forward with each new snapshot.

Whilst at first glance appearing somewhat complicated, the approach above will allow participants to choose what subset of the release points of each versioned element they implement, and does not restrict those participants who wish to aggressively advance their IT infrastructure.

From an application perspective, the key is thus the selection of containers to carry version information.

3.2.8 Selecting type of versioning

A range of versioning from implicit (utilising defaults) to explicit (Mandatory) can be employed in the definition of any XML schema.

Rather than provide a “one size fits all” approach to versioning it is recommended that due consideration be given to the level of flexibility that is required when defining the versioning approach within the schema

The level to which versioning within the schema needs to be defined and enforced can vary depending upon the requirements of the market / system that need to use aseXML. See section 3.22 and process messages. This needs to be considered carefully when the transactions are initially defined as it is unlikely that the manner in which they are used will be able to be changed due to the dependencies that are established upon them. The application of versioning can be any of the following depending upon the need.

Where the utilisation of versioning is seen to be of little importance or a significant amount of diversity is desired to be accommodated then an approach to versioning should tend to be more implicit. If a highly prescriptive application of versioning is required then more explicit versioning may be required. These need to be considered carefully when the transactions are initially defined as it is unlikely that the manner in which they are used will be able to be changed due to the system dependencies that can be built around them.

Consideration needs to be given also to the type of change being implemented. Notwithstanding the guiding principles, see 3.2.1, where a change is extending current functionality then the type of versioning employed should be consistent with what is already in place. i.e. if Mandatory versioning is already in place then this should be utilised. If new functionality is being introduced then whatever versioning is appropriate can be used.

Version tags attributes can be defined as with the following attributes:

Mandatory

As the name suggests this is a highly prescriptive way of using versioning information. The schema specifies what the valid version is and the instance document must explicitly carry the version information as defined by the validating schema. It essentially provides information, separate from the namespace, within the instance document that can be utilised by systems. This type of versioning may be used where “precision” about what is being delivered is needed, see section 3.2.3. Where changes to the schema are adopted, dependent systems will need to be modified to accommodate the changes in versioning. would be used where the architecture of the IT systems or the business processes require it.

Optional

In this case, as the name implies, it is optional that the version information be supplied in the instance document. Thus when adopting a new schema this would mean that no changes need to be made, apart from the namespace where the content, in this case of documents, stay the same.

Optional with Default

The schema contains as a default what the version attribute is. Where this versioning is utilised it can be used by the recipient or sender of the document to determine what the version is if it's not supplied in the instance document. i.e. a validating parser can populate the document with the version information and the output file can be used if that is needed.

3.2.9 Selecting Elements To Version

In line with section 3.2.1 point 1, the number of elements carrying version attributes needs to be limited to a manageable level.

Version attributes are to be added to an element when it is considered of sufficient importance from an application perspective to warrant it.

It should be noted that where a versioned element is contained by another versioned element, it is quite possible for the contained element to carry a more recent release identifier than the containing element (see section 3.4.4).

The following rules may be used in assessing the need for a version attribute on a particular element.

1. Is the element important in the overall framework of aseXML?

This rule has been applied to mandate the provision of a version attribute on <Transaction> sub-elements (see section 9.3).

It would also allow future addition of version attributes to the aseXML <Header> element (see section 9.2) or to the <aseXML> element (see section 9.1) itself.

2. Is the element a significant data structure shared across multiple transactions?

The aim here is to localise a change that would otherwise have a significant version ripple effect across multiple transactions within aseXML. The assumption is that a common piece of code may be used to handle the structure and isolate any version differences from calling code.

Localisation needs to be balanced against the desire to limit the number of version attributes to a manageable level.

For instance, it may be more appropriate to change the version number of several transactions rather than introduce a version attribute to a shared element. If in doubt, the decision should be biased towards minimising the number of versioned elements.

This rule would, for instance, allow the introduction of a version attribute on the <Event> element in the case of a change to its structure.

It is recommended that new significant structure definitions considered likely candidates for future updates carry an initial version attribute for consistency.

3. Is the element a container for multiple different formats, each of which needs to be separately versioned?

This rule has been applied to support the inclusion of a version attribute on any changes to concrete derivations of abstract base types used to define elements within aseXML.

Note that in this case, the version attribute is not associated directly with the abstract element; rather it is part of the data definition of the derived concrete type.

As an example, any changes to NMI Standing Data or table replication formats (see section 12.1) will result in the new definition carrying a version attribute.

It is recommended that new concrete derivations (as distinct from modifications to existing definitions) carry an initial version attribute for consistency.

Because this recommendation was not in force from the creation of aseXML, aseXML schemas may contain some concrete type definitions that do not include a version attribute.

3.2.10 “Backwards compatible” changes

There are a number of circumstances in which application code may be able to handle changes in the XML structure and content without needing to refer to version information. For instance, code changed to accommodate an increase in the length of a text element will inherently handle any previous document that provided a value of shorter length.

The question this example raises is whether aseXML should distinguish between

- changes that require the code to know which version of an element they are dealing with, and
- changes that can be accommodated by common code logic.

This distinction is often referred to in terms of “versions” vs “revisions” or “major” vs “minor” versions.

In the interests of simplicity and consistency, the aseXML versioning model requires that ANY change to the schema that results in a change to the structure or content of an aseXML fragment requires a change in the version attribute of the nearest enclosing versioned element. (I.e. if part of a related aseXML instance document under a version attribute changes such that it could affect the processing or structure of the document) Conversely, this means that if the aseXML fragment doesn't change then the version attribute does not need to be updated.

Despite the resulting increased rate of change of version information, application code is free to ignore the version information, or only trigger on a subset of the release points.

The rules for modifying version information are thus simple to grasp and easy to apply for any change scenario.

3.3 USING DEVELOPMENT IDENTIFIERS

This section presents a scenario to demonstrate the use of development release identifiers in moving from one production release to the next.

3.3.1 Scenario

aseXML is at production release r100. It becomes evident that a new production release is needed as a result of changes to the operation of the market. An element in transaction T1 must be updated, the type definition of which is also used by an element in T2.

Two organisations (A and B) agree to take the lead in development of the change. The sequence of events is detailed in the next section and shown diagrammatically in Figure 2.

3.3.2 Sequence of Events

1. The letter “a” is assigned to the development thread. In this case, a type definition used by an element in T1 must change. The type definition is also used by transaction T2. Transaction T3 is not affected by the change. Because the affected element doesn’t carry a version attribute, the transaction element definitions represent the nearest enclosing versioned element and thus a new release point must be used for T1 and T2.
2. A and B communicate privately and decide upon a first cut of the changes.
3. A copy of the current production release of the aseXML schemas is taken.
4. A and B agree on a development extension. A and B choose r100_a1.
5. All references to the namespace in the schema files are updated to the development release, together with the version identifiers within the affected schema filenames. The version attributes of the affected transactions are updated to contain the development release identifier.
6. A and B enhance their infrastructure to support the changes. There may be multiple iterations and depending on the schema infrastructure used, the development release identifier may change as agreed by A and B.
7. A and B are ready for interoperability testing and feel the change is ready for public scrutiny.

8. An area within the web site containing the aseXML schemas is created for the development release and the complete schema is placed on the site.
9. As a result of testing between A and B and public comments, steps 4 to 8 may be repeated.
10. Agreement is reached between A and B that the change is a candidate for production release. Checks are carried out to integrate any changes as a result of other completed development threads.
11. A period is entered during which other organisations who choose to enhance their infrastructure in parallel to A and B may now request A and/or B to provide conformance testing of their implementation.
12. Agreement is reached amongst participants that the change is ready for production release.
13. The next production release is assigned and step 5 performed using the production release. In this scenario, the new release is r101.

It would however have been possible that r101 was released as a result of a different development thread. According to the process above, the changes in r101 would need to be rolled into the development thread and the production release would be r102.
14. An area within the web site containing the aseXML schemas is created for the production release and the complete schema is placed on the site. The files for all development releases of this thread are deleted from the site.
15. Other participants may now implement this release point of the transactions according to their IT schedules.

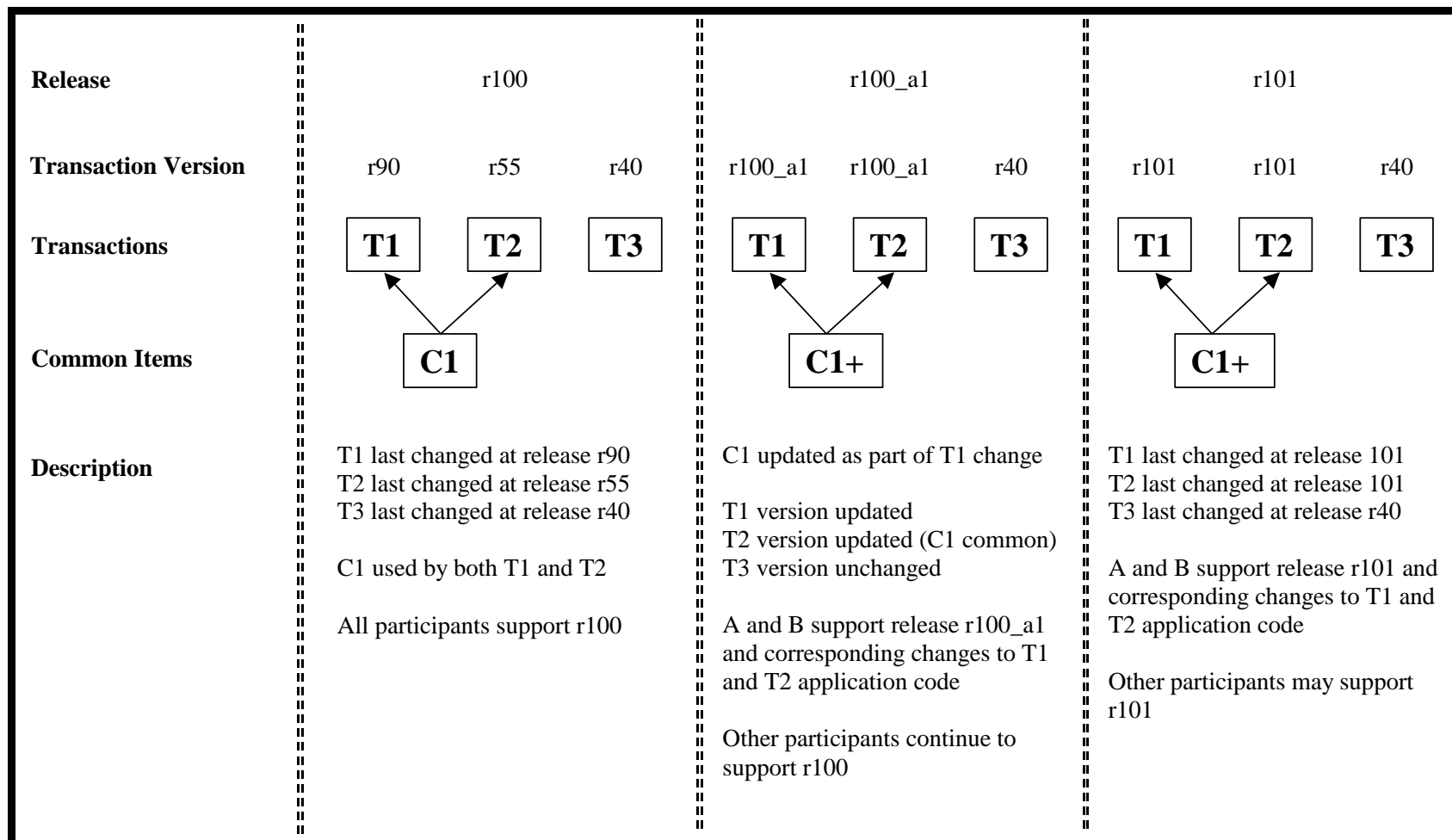


Figure 2 – Introducing a Change To aseXML

3.4 ARCHITECTURE IMPLICATIONS OF aseXML RELEASE MIGRATIONS

The version scheme of aseXML has been designed to allow as smooth a transition as possible from one release to the next. It recognises that synchronisation of IT development cycles across multiple organisations is difficult to achieve and tries, where possible, to allow software upgrades to occur independently.

This section suggests a number of architectural features that will further minimise the impact of schema migrations on IT infrastructures and project timings.

3.4.1 Accepting aseXML Messages

- Support Multiple Schemas

For many organisations, the changes in a new release may not affect any of the transactions they support.

The only difference in accepted instance documents between the previous and new release will be the namespace of the top-level element. The structure and version information of the accepted transactions remains the same and thus represents no change to application logic.

Architectures should thus support the ability to accept transactions under a number of different schema releases. This allows other organisations to move their produced messages to a later schema release without significantly impacting those organisations for which the changes are of no interest.

The version scheme assumes that participants will accept new schema releases as they become available even though the changes may not affect their infrastructure.

- Isolate Validity Checking From Handler Selection

The decision as to which handler to invoke for a particular transaction should be based on a combination of the transaction group, transaction name and version attribute of the transaction itself.

The namespace information should only be used in the process of validating the message.

This allows the message to be validated under multiple schemas without needing to change the handler selection logic each time a new namespace is produced.

- Simple schema installation

It is important that the process of adding support for validation of instance documents against a new schema should be as simple as possible. Depending on the way in which the mapping of namespace to schema is performed, this may be as simple as placement of the schema files in the appropriate directory.

3.4.2 Producing aseXML Messages

- Selecting The Message Namespace

The namespace used on outgoing messages need only change when the release point of one of the produced elements changes. It is possible for the release on output to lag the release on input.

The output release should thus be decoupled from the accepted input releases.

- Simple Control Of Namespace/SchemaLocation

Over time, the number of schema releases in use may grow to the point where it is appropriate to remove one or more of the older releases from use. If the output release being used is one of these older releases, migrating to a newer release on output may simply involve a change of namespace.

The process of controlling the namespace information in a produced aseXML message should thus be as simple as possible.

3.4.3 Minimising Code Branches

It is generally accepted in software development that where possible, reducing the number of logic branches in code enhances its maintainability.

While aseXML provides version attributes to facilitate such branches, where possible updates to versioned elements should be designed such that a single code module can handle the variability caused by the various versions without reference to the version information. The points below suggest ways in which this can be achieved. The focus is on extending rather than restricting the contents of transactions.

- Optional Elements/Attributes

Where possible, new elements/attributes added to trigger additional business logic should be made optional. This allows for the possibility that the recipient may not be interested in the additional functionality, while allowing the producer to move their infrastructure forward.

- Ignore Unknown Elements/Attributes

Code should be designed to ignore unknown or unexpected elements/attributes rather than flag these as an error. This is a natural extension of the use of optionality but may apply to mandatory elements/attributes in some circumstances.

- Extend Content Models

For data types such as enumerations, removal of values should be avoided where possible since it automatically invalidates existing code producing the element/attribute concerned.

- Handle Unknown Values

Default handling of unrecognised values should be employed and, where possible, not result in errors or transaction rejections.

3.4.4 Release Example

The table below shows a possible sequence of releases affecting a set of versioned elements. T4 and T5 share a versioned element E1. The table contents show the release point for each versioned element under each release.

Versioned Element	Release				
	r1	r2	r3	r4	r5
T1	r1	r2	r2	r4	r4
T2	r1	r1	r3	r3	r3
T3	r1	r1	r1	r1	r1
E1			r3	r3	r3
T4(E1)	r1	r1	r1	r1	r5
T5(E1)	r1	r1	r1	r4	r5

A few points should be noted from the above example.

1. T3 does not change across all the releases and could be delivered under any of them. An organisation only interested in producing T3 could leave the release of their produced messages at r1.
2. T1 changed at release r2 and r4. T2 changed at release r3.
3. E1 initially did not have a version attribute, but at release r3 was considered of sufficient importance to have one introduced. The release point of T4 and T5 at release r3 thus did not change, since the version attribute on E1 limited the effect of the change. The effect of

this was that the release point of the contained element (E1) was more recent than that of the containing element (T4, T5).

4. At release r4, T5 was altered. Because the release point of E1 remained unchanged, the changes were either outside the E1 element, or were changes to the use of E1 in T4, e.g. its optionality or cardinality.
5. At release r5, both T4 and T5 were altered, but E1 again remained unchanged.

3.5 aseXML VERSIONING STEP BY STEP

This section presents a sequence of changes to the allowable content and structure of a sample aseXML message. Further information on the exact structure of aseXML messages may be found in subsequent chapters.

For each change, the effects on the version information are discussed. Affected elements for each step are highlighted in bold in the XML.

3.5.1 The Initial Message

```
<?xml version="1.0"?>
<ase:aseXML xmlns:ase="urn:aseXML:r7"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:aseXML:r7
http://www.nemmco.com.au/aseXML/schemas/r7/aseXML_r7.xsd">
  <Header>
    <EH1>EH1 Text</EH1>
    <EH2>EH2 Text</EH2>
  </Header>
  <Transactions>
    <Transaction>
      <T1 version="r3">
        <E1>4.32</E1>
        <AbstractE1 xsi:type="Type1">
          <EA1>EA1 String</EA1>
        </AbstractE1>
      </T1>
    </Transaction>
    <Transaction>
      <T2 version="r4">
        <EA1>E2 String</EA1>
        <E3>
          <E3a>E3a text</E3a>
        </E3>
        <E4>1.2</E4>
      </T2>
    </Transaction>
    <Transaction>
      <T3 version="r4">
        <E5>5050</E5>
        <E6>2001-05-01</E6>
        <AbstractE2 xsi:type="Type2">
          <EA2>EA1 String</EA2>
          <EA3>1999-01-01</EA3>
        </AbstractE2>
      </T3>
    </Transaction>
    <Transaction>
      <T4 version="r6">
        <E4>4.32</E4>
        <AbstractE2 xsi:type="Type2">
          <EA4>EA3 String</EA4>
        </AbstractE2>
      </T4>
    </Transaction>
  </Transactions>
</ase:aseXML>
```

The points below should be noted with regard to this message.

- The message uses release r7.
- The release points of versioned elements within the message will not in general match that of the message.
- <T3> and <T4> use a common element, <AbstractE2>, whose actual contents are validated according to the value of the `xsi:type` attribute.

3.5.2 The Content Model Changes For An Isolated Element

```
<?xml version="1.0"?>
<ase:aseXML xmlns:ase="urn:aseXML:r8"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:aseXML:r8
http://www.nemmco.com.au/aseXML/schemas/r8/aseXML_r8.xsd">
  <Header>
    <EH1>EH1 Text</EH1>
    <EH2>EH2 Text</EH2>
  </Header>
  <Transactions>
    <Transaction>
      <T1 version="r8">
        <E1>4.3</E1>
        <AbstractE1 xsi:type="Type1">
          <EA1>EA1 String</EA1>
        </AbstractE1>
      </T1>
    </Transaction>
    <Transaction>
      <T2 version="r4">
        <EA1>E2 String</EA1>
        <E3>
          <E3a>E3a text</E3a>
        </E3>
        <E4>1.2</E4>
      </T2>
    </Transaction>
    <Transaction>
      <T3 version="r4">
        <E5>5050</E5>
        <E6>2001-05-01</E6>
        <AbstractE2 xsi:type="Type2">
          <EA2>EA1 String</EA2>
          <EA3>1999-01-01</EA3>
        </AbstractE2>
      </T3>
    </Transaction>
    <Transaction>
      <T4 version="r6">
        <E4>4.32</E4>
        <AbstractE2 xsi:type="Type2">
          <EA4>EA3 String</EA4>
        </AbstractE2>
      </T4>
    </Transaction>
  </Transactions>
</ase:aseXML>
```

The points below should be noted with regard to this message.

- The content model for element <E1> was altered to limit it to one decimal place.
- <T1> is the nearest enclosing versioned element for <E1>, so it takes on a new release point of r8.
- Since <E1> was only used by <T1>, only the version information of <T1> was affected.

3.5.3 The Content Model Changes For A Shared Element

```
<?xml version="1.0"?>
<ase:aseXML xmlns:ase="urn:aseXML:r9"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:aseXML:r9
http://www.nemmco.com.au/aseXML/schemas/r9/aseXML_r9.xsd">
  <Header>
    <EH1>EH1 Text</EH1>
    <EH2>EH2 Text</EH2>
  </Header>
  <Transactions>
    <Transaction>
      <T1 version="r8">
        <E1>4.3</E1>
        <AbstractE1 xsi:type="Type1">
          <EA1>EA1 String</EA1>
        </AbstractE1>
      </T1>
    </Transaction>
    <Transaction>
      <T2 version="r9">
        <EA1>E2 String</EA1>
        <E3>
          <E3a>E3a text</E3a>
        </E3>
        <E4>A string</E4>
      </T2>
    </Transaction>
    <Transaction>
      <T3 version="r4">
        <E5>5050</E5>
        <E6>2001-05-01</E6>
        <AbstractE2 xsi:type="Type2">
          <EA2>EA1 String</EA2>
          <EA3>1999-01-01</EA3>
        </AbstractE2>
      </T3>
    </Transaction>
    <Transaction>
      <T4 version="r9">
        <E4>Another string</E4>
        <AbstractE2 xsi:type="Type2">
          <EA4>EA3 String</EA4>
        </AbstractE2>
      </T4>
    </Transaction>
  </Transactions>
</ase:aseXML>
```

The points below should be noted with regard to this message.

- The content model for element <E4> was altered from a decimal to a string.
- <T2> and <T4> are the nearest enclosing versioned elements for <E4>, so both take on a new release point of r9.

3.5.4 The Content Model Changes For A Shared Element (2)

```
<?xml version="1.0"?>
<ase:aseXML xmlns:ase="urn:aseXML:r10"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:aseXML:r10
http://www.nemmco.com.au/aseXML/schemas/r10/aseXML_r10.xsd">
  <Header>
    <EH1>EH1 Text</EH1>
    <EH2>EH2 Text</EH2>
  </Header>
  <Transactions>
    <Transaction>
      <T1 version="r8">
        <E1>4.3</E1>
        <AbstractE1 xsi:type="Type1" version="r10">
          <EA1>19.2</EA1>
        </AbstractE1>
      </T1>
    </Transaction>
    <Transaction>
      <T2 version="r10">
        <EA1>2.4</EA1>
        <E3>
          <E3a>E3a text</E3a>
        </E3>
        <E4>A string</E4>
      </T2>
    </Transaction>
    <Transaction>
      <T3 version="r4">
        <E5>5050</E5>
        <E6>2001-05-01</E6>
        <AbstractE2 xsi:type="Type2">
          <EA2>EA1 String</EA2>
          <EA3>1999-01-01</EA3>
        </AbstractE2>
      </T3>
    </Transaction>
    <Transaction>
      <T4 version="r9">
        <E4>Another string</E4>
        <AbstractE2 xsi:type="Type2">
          <EA4>EA3 String</EA4>
        </AbstractE2>
      </T4>
    </Transaction>
  </Transactions>
</ase:aseXML>
```

The points below should be noted with regard to this message.

- The content model for element <EA1> was altered from a string to a decimal.
- In the case of <T1>, because <EA1> is within a concrete derivation of an abstract type, a version attribute is added to Type1. The release point of <T1> thus remains unchanged.
- In the case of <T2>, the <T2> element is the nearest versioned element and so it moves to the r10 release point.

3.5.5 The Structure Changes For A Shared Concrete Type

```
<?xml version="1.0"?>
<ase:aseXML xmlns:ase="urn:aseXML:r11"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:aseXML:r11
http://www.nemmco.com.au/aseXML/schemas/r11/aseXML_r11.xsd">
  <Header>
    <EH1>EH1 Text</EH1>
    <EH2>EH2 Text</EH2>
  </Header>
  <Transactions>
    <Transaction>
      <T1 version="r8">
        <E1>4.3</E1>
        <AbstractE1 xsi:type="Type1" version="r10">
          <EA1>19.2</EA1>
        </AbstractE1>
      </T1>
    </Transaction>
    <Transaction>
      <T2 version="r10">
        <EA1>2.4</EA1>
        <E3>
          <E3a>E3a text</E3a>
        </E3>
        <E4>A string</E4>
      </T2>
    </Transaction>
    <Transaction>
      <T3 version="r4">
        <E5>5050</E5>
        <E6>2001-05-01</E6>
        <AbstractE2 xsi:type="Type2" version="r11">
          <EA2>EA1 String</EA2>
          <EA3>1999-01-01</EA3>
          <EA5>true</EA5>
        </AbstractE2>
      </T3>
    </Transaction>
    <Transaction>
      <T4 version="r9">
        <E4>Another string</E4>
        <AbstractE2 xsi:type="Type2" version="r11">
          <EA4>EA3 String</EA4>
          <EA5>false</EA5>
        </AbstractE2>
      </T4>
    </Transaction>
  </Transactions>
</ase:aseXML>
```

The points below should be noted with regard to this message.

- An additional element is added to the end of the definition of concrete derivation Type2. As a result, a version attribute is added.
- The release point of <T3> and <T4> remain unchanged, since the version attribute on Type2 limits the effect of the change.

3.5.6 The Optionality Of An Element is Changed

```
<?xml version="1.0"?>
<ase:aseXML xmlns:ase="urn:aseXML:r12"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:aseXML:r12
http://www.nemmco.com.au/aseXML/schemas/r12/aseXML_r12.xsd">
  <Header>
    <EH1>EH1 Text</EH1>
    <EH2>EH2 Text</EH2>
  </Header>
  <Transactions>
    <Transaction>
      <T1 version="r8">
        <E1>4.3</E1>
        <AbstractE1 xsi:type="Type1" version="r10">
          <EA1>19.2</EA1>
        </AbstractE1>
      </T1>
    </Transaction>
    <Transaction>
      <T2 version="r12">
        <EA1>2.4</EA1>
        <E3>
          <E3a>E3a text</E3a>
        </E3>
        <E4>A string</E4>
      </T2>
    </Transaction>
    <Transaction>
      <T3 version="r4">
        <E5>5050</E5>
        <E6>2001-05-01</E6>
        <AbstractE2 xsi:type="Type2" version="r11">
          <EA2>EA1 String</EA2>
          <EA3>1999-01-01</EA3>
          <EA5>true</EA5>
        </AbstractE2>
      </T3>
    </Transaction>
    <Transaction>
      <T4 version="r9">
        <E4>Another string</E4>
        <AbstractE2 xsi:type="Type2" version="r11">
          <EA4>EA3 String</EA4>
          <EA5>false</EA5>
        </AbstractE2>
      </T4>
    </Transaction>
  </Transactions>
</ase:aseXML>
```

The points below should be noted with regard to this message.

- The <E4> element in <T2> is changed from being optional to mandatory.
- The release point of <T2> changes to reflect the change in optionality of <E4>.
- While <T4> also contains <E4>, the use of <E4> in <T4> remains unchanged and hence there is no change in its release point.

3.5.7 Version Attribute Added

```
<?xml version="1.0"?>
<ase:aseXML xmlns:ase="urn:aseXML:r13"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:aseXML:r13
http://www.nemmco.com.au/aseXML/schemas/r13/aseXML_r13.xsd">
  <Header version="r13">
    <EH1>EH1 Text</EH1>
    <EH2>EH2 Text</EH2>
    <EH3>false</EH3>
  </Header>
  <Transactions>
    <Transaction>
      <T1 version="r8">
        <E1>4.3</E1>
        <AbstractE1 xsi:type="Type1" version="r10">
          <EA1>19.2</EA1>
        </AbstractE1>
      </T1>
    </Transaction>
    <Transaction>
      <T2 version="r12">
        <EA1>2.4</EA1>
        <E3>
          <E3a>E3a text</E3a>
        </E3>
        <E4>A string</E4>
      </T2>
    </Transaction>
    <Transaction>
      <T3 version="r4">
        <E5>5050</E5>
        <E6>2001-05-01</E6>
        <AbstractE2 xsi:type="Type2" version="r11">
          <EA2>EA1 String</EA2>
          <EA3>1999-01-01</EA3>
          <EA5>true</EA5>
        </AbstractE2>
      </T3>
    </Transaction>
    <Transaction>
      <T4 version="r9">
        <E4>Another string</E4>
        <AbstractE2 xsi:type="Type2" version="r11">
          <EA4>EA3 String</EA4>
          <EA5>false</EA5>
        </AbstractE2>
      </T4>
    </Transaction>
  </Transactions>
</ase:aseXML>
```

The points below should be noted with regard to this message.

- The <EH3> element is added to <Header>.
- Since there is no enclosing versioned element, a version attribute is added to <Header>.

4. NAMESPACES

4.1 aseXML NAMESPACE FORMAT

The aseXML namespace name will use a URN (see RFC 2396) of the format shown below,

urn:aseXML:ReleasIdentifier

where

- **ReleasIdentifier is the release identifier of the namespace as per section 3.2.5.**

Thus an example of the aseXML namespace might be

urn:aseXML:r100

4.2 DEFAULT NAMESPACES

The XML Namespace specification allows the use of a default namespace to simplify, in some cases, the need to identify what elements come from what namespace.

Schemas for aseXML should use a default namespace matching the targetNamespace. For schemas not specifying a targetNamespace, no default namespace should be defined.

Instance documents should not use a default namespace due to the element qualification style being used (see section 6.7). Rather, they should qualify the root element with the appropriate aseXML namespace.

4.3 NAMESPACE PREFIXES

The case-sensitive namespace prefixes in the table below will be used in schemas and instance documents.

Namespace	Prefix
<u>World Wide Web Consortium</u>	
http://www.w3.org/2001/XMLSchema	xsd
http://www.w3.org/2001/XMLSchema-instance	xsi
http://www.w3.org/1999/XSL/Transform	xsl
http://www.w3.org/1999/XSL/Format	fo
<u>aseXML</u>	
urn:aseXML:r?	ase

5. SCHEMA ORGANISATION

5.1 SCHEMALOCATION URLs

As per the guidelines discussed in section 3.2.6, schemas need to be able to be fetched from the web via HTTP. In addition, the generation of a schemaLocation attribute for a given namespace should be able to be automated.

It is envisaged that initially NEMMCO will host the aseXML schema files, but that in the longer term, a more suitable location may be found.

Given that the schemaLocation attribute may contain more than one namespace/URI pair for a single namespace, such a move is easily accommodated.

The format of a URL for use in schemaLocation attributes is shown below;

WebSiteRoot/schemas/ReleasIdentifier/aseXML_ReleasIdentifier.xsd

where

- **WebSiteRoot is the root portion of the URL needed to gain access to the web site.**
- **ReleasIdentifier is that of the corresponding namespace and complying with section 3.2.5.**

Thus, an example of a URL might be

http://www.nemmco.com.au/aseXML/schemas/r100/aseXML_r100.xsd

The ReleasIdentifier is included in the filename portion of the URL so that the filename remains unique even when separated from the rest of the URL, for instance in a local parser cache. The ReleasIdentifier is also included in the URL path in line with section 3.2.6.

All resources under a given ReleasIdentifier directory will carry a ReleasIdentifier as the last part of the filename prior to the extension.

The ReleasIdentifier on each file will reflect the release at which the file last changed. This will assist in identifying the set of schema files affected by a particular release. A side effect of this is that the same schema file may appear in the directories of multiple releases.

All schema files will use a .xsd extension.

5.2 TRANSACTION FILES

In order to improve the maintainability of the aseXML schemas, multiple files will be used to hold the schema for a particular release of aseXML. These files will be included into the schema identified by section 5.1 via the XML Schema include mechanism (see section 5.3).

A file may contain all transaction exchanges for an application, a single transaction exchange within an application or one transaction within a transaction exchange. The choice is left to the developer, with the overriding principle being to minimise the number of files used.

In the case of a single file per application, the filename will take the form

ApplicationTitle_ReleasIdentifier.xsd

where

- **ApplicationTitle** is replaced with the short title of the application. It may contain alphanumeric characters and will use title case.
- **ReleasIdentifier** is that of the corresponding namespace and will comply with section 3.2.5.

An example of such a file might be

NMIDataAccess_r100.xsd

In the case of a single file per transaction exchange, the filename will take the form

ExchangeTitle_ReleasIdentifier.xsd

where

- **ExchangeTitle** is replaced with the short title of the transaction exchange. It may contain alphanumeric characters and will use title case.
- **ReleasIdentifier** is that of the corresponding namespace and will comply with section 3.2.5.

An example of such a file might be

NMIDiscovery_r100.xsd

In the case of a file per transaction of a transaction exchange, the filenames will take the form

ExchangeTitleTransactionDescription_ReleaseIdentifier.xsd

where

- **ExchangeTitle** is replaced with the short title of the transaction exchange as above.
- **TransactionDescription** is replaced with the short title of the transaction in question. It may contain alphanumeric characters and will use title case. In the common case of a single, two-way exchange, the texts “Request” and “Response” will be used.
- **ReleaseIdentifier** is that of the corresponding namespace and will comply with section 3.2.5.

An example of the files in this case might be

NMIDiscoveryRequest_r100.xsd

NMIDiscoveryResponse_r100.xsd

5.3 SCHEMA INCLUSION

Where schemas are included in other schemas via an `<include>` element, only relative URLs will be used consisting of the filename only.

An example of an include element within a schema is given below.

```
<include schemaLocation="NMIDiscovery_r100.xsd"/>
```

The included schema should NOT have a `targetNamespace` attribute and should not use a default namespace, in accordance with section 1.5, reference 2.

5.4 COMMON SCHEMAS

As a minimum, the type definitions common across multiple transactions will be split across three files as shown in the table below. See section 6.5 for a discussion of abstract types.

Where a group of common definitions logically stands alone, these should be placed in their own schema file. An example of this might be type definitions for addresses.

Schema File	Usage
Common_r?.xsd	Concrete definitions for common types Abstract definitions for fuel specific variants (see section 6.5)
Gas_r?.xsd	Concrete derivations for gas of abstract types Gas specific type definitions
Electricity_r?.xsd	Concrete derivations for electricity of abstract types Electricity specific type definitions

5.5 ELEMENTS/TYPES

Element and type names will use title case and alphanumeric characters.

An example might be

StreetName

Plural names should only be used for collections, typically where repeating sub-elements are expected.

Element/type names should be kept to 40 characters in length.

Where acronyms cause two upper case characters to be adjacent, they may be separated by an underscore to improve clarity.

An example might be

PO_Box

Where possible, an element name and its corresponding type name should be identical.

5.6 TRANSACTION ELEMENTS

The names used for elements representing each transaction will take the form

ExchangeTitleTransactionDescription

where

- **ExchangeTitle is replaced with the short title of the transaction exchange as in section 5.2.**

- **TransactionDescription is replaced with the short title of the transaction as in section 5.2.**

Examples of elements might be

NMIDiscoveryRequest

NMIDiscoveryResponse

There will be a type per transaction allowing them to be individually checked against a schema. The type and element will use the same name as per section 5.5.

5.7 ATTRIBUTES

Attribute names will use title case and alphanumeric characters with the first letter lowercase.

An example might be

version

This is in keeping with the formatting used in the XML standards (c.f. schemaLocation).

Attribute names should be kept to 25 characters in length.

6. SCHEMA FEATURES

6.1 XML DECLARATION

All schemas will include an XML declaration.

An example is shown below.

```
<?xml version="1.0" ?>
```

The default encoding of UTF-8 is assumed. All XML implementations must support UTF-8 to comply with the Extensible Markup Language (XML) 1.0 specification, with the ASCII character set being a subset of it.

6.2 ANONYMOUS vs NAMED TYPES AND DATA DICTIONARIES

The XML Schema standard allows for types to be defined in-line at their point of use (anonymous types) or to be named explicitly. Whilst the former approach leads to more compact definitions, it makes the automated production of data dictionaries from the schemas more difficult. Additional information with regards to a type more logically resides with an explicit definition of the type, rather than embedded within a transaction.

As a result, authors are encouraged to define named types for data items and item groups.

6.3 ANNOTATIONS

Annotations allow association of comments with arbitrary elements within a schema and provide a way to make schemas somewhat self-documenting. Tools such as XMLSpy display these comments when creating XML documents from the schemas.

The use of annotations is encouraged within aseXML schemas.

As a minimum, each schema file and type/element definition will include an annotation containing a brief description of its purpose.

For transaction elements, the description should include the TransactionGroup to which the transaction belongs (see section 9.2.4).

The definition of the annotation element is such that it allows user defined content in terms of other markup. To further facilitate the automatic production of data dictionaries, three sub-elements of the documentation element are recommended;

<ChangeHistory> - documents what has been changed

<DeveloperNotes> - documents why changes were made

<UsageNotes> - information to assist the creators of aseXML compliant transactions

6.4 SIMPLE TYPES

In order to maximise the value of the schema in validating instance documents, simple types will be designed to be as restrictive as possible. This is achieved by the use of the facets facility within XML Schemas.

By preference, the enumeration facet should be used where possible, as discussed in section 2.4.

6.5 HANDLING FUEL SPECIFIC VARIATIONS

In order to accommodate multiple fuels within the aseXML transactions, it will be necessary to allow for element variants. The aim should be to minimise any duplication and maximise the parser's ability to reject invalid document instances.

XML schemas provide two mechanisms by which variants might be achieved – choice elements and type derivation by extension.

Choice elements allow one of a number of elements to appear at a given location in a document instance. The advantage of this approach is that the name of the included element clearly indicates its semantics. A choice between multiple groups of elements is also possible.

Type derivation by extension follows the classical object-oriented paradigm where the derived types may be used anywhere that the base type appears in a schema. In addition, the base type may be declared as abstract forcing only the derived types to be valid in an instance document. In order to assist the parser in determining the appropriate type, instance documents must provide the `xsi:type` attribute on elements of the derived types. Abstract definitions are only supported on complex types.

By preference, type derivation by extension from an abstract base type should be used to resolve fuel variants. The base type will be defined in the `Common_r?.xsd` file with the `abstract` attribute set to `true`. The fuel specific variants should be defined in the appropriate fuel type file. Use of abstract types will allow commonality of transactions across fuels whilst collecting the fuel specific variants in a common location.

Where there is little commonality between fuel variants, or where simple types are involved, use of a choice may be preferable. Use of choice statements leads to simpler instance documents but has the disadvantage that the choice statement must appear in the schema wherever the choice between fuel variants is required.

The `xsi:type` attribute will allow applications to easily detect which fuel type is involved.

6.6 aseXML ATTRIBUTES

Where attributes are defined for aseXML elements, the issues below should be considered.

6.6.1 Default Values

In order to make instance documents as self-explanatory as possible, it is desirable that attribute definitions in aseXML schemas force the inclusion of the attribute in all instance documents.

6.6.2 ID And IDREF

Where ID and IDREF attributes are used to provide linkage between elements, the ID value used need only be unique to the document instance with no requirement for global uniqueness.

6.7 ELEMENT AND ATTRIBUTE QUALIFICATION

Both elements and attributes will use the default values for namespace qualification, i.e. "unqualified". Only top-level elements in instance documents will need to be qualified with the version of the namespace name corresponding to the release point of the transactions.

6.8 HANDLING EMPTY STRING ELEMENTS

Mandatory or optional string elements can not be empty in the instance documents. If the string elements are not restricted by enumerations or patterns, they should be restricted by minimum length of one and by disallowing a white space.

This will be achieved by use of the nonEmptyString simple type in the construct of the mandatory string element.

The nonEmptyString will have the following structure:

```
<xsd:simpleType name="nonEmptyString">  
  <xsd:restriction base="xsd:string">  
    <xsd:minLength value="1"/>  
    <xsd:whiteSpace value="collapse"/>  
  </xsd:restriction>  
</xsd:simpleType>
```

NonEmptyString is a string restricted by minimum length of one character. This character can not be a 'whitespace'.

Minimum length of one character disallows provision of an empty string while 'whitespace' restriction disallows provision of 'whitespace' in that character.

'Whitespace' characters are the following characters:

- ' ' blank
- '\t' tab
- '\n' new line
- '\f' form feed

'\r' carriage return

For additional explanations of the 'whitespace' term please see [http://en.wikipedia.org/wiki/Whitespace_\(computer_science\)](http://en.wikipedia.org/wiki/Whitespace_(computer_science)).

Here are some examples of how the nonEmptyString can be used:

1. Author string element

```
<xsd:complexType name="book">
  <xsd:sequence>
    <xsd:element name="author" type="nonEmptyString"
      (minOccurs="0")/>
    <xsd:element name="price" type="xsd:positiveInteger"/>
  </xsd:sequence>
</xsd:complexType>
```

The following XML files will **NOT** be valid:

```
<book>
  <author></author>
  <price>20</price>
</book>

<book>
  <author> </author>
  <price>20</price>
</book>
```

The following XML files will be valid:

```
<book>
  <author>Book Author</author>
  <price>20</price>
</book>
```

6.8.1 Nillable string element

If the intention is deliberately to provide a string element which is empty than **nillable="true"** attribute should be used:

Here is an example of how **nillable="true"** can be used along with the "nonEmptyString".

```
<xsd:complexType name="book">
  <xsd:sequence>
    <xsd:element name="author" type="nonEmptyString"
      (minOccurs="0") nillable="true"/>
    <xsd:element name="price" type="xsd:positiveInteger"/>
  </xsd:sequence>
</xsd:complexType>
```

In this example, a deliberate intention to provide an empty “author” can be achieved by `nillable="true"` attribute:

```
<author xsi:nill="true"/>  
<price>20</price>
```

7. INSTANCE DOCUMENTS

7.1 XML DECLARATION

All instance documents will include an XML declaration identical to that of the schemas.

7.2 DEFAULT NAMESPACES

Default namespaces will not be used in instance documents, due to the qualification style being used (see section 6.7). Top-level elements should be explicitly prefixed with “ase” as per section 4.3.

7.3 SCHEMALOCATION ATTRIBUTE

Whenever an aseXML namespace is declared, the corresponding xsi:schemaLocation attribute should be included in the instance document. Refer to section 5.1 for details.

7.4 DECLARING NAMESPACES FROM THE XML STANDARDS

Declarations for namespaces such as the XML Schema Instance namespace will occur on the top-level element of any instance document. The prefixes used will follow section 4.3.

8. TRANSPORT, ENVELOPE OR TRANSACTION

In order to clearly identify what it is that needs to be specified as part of producing the transactions for a given application, a distinction needs to be made between the XML defined for each transaction and the XML needed to carry information about the transaction.

Figure 3 presents a high level logical view of the IT framework needed by a participant to handle aseXML transactions. This is a simplification of the XML stack presented in the white paper with all layers below the envelope collapsed into the Transport layer.

In this model, it is the responsibility of the Transport/Envelope layers to provide the meta-information about the transaction. Once the XML for these layers is standardised, developers of a process need only consider the XML needed at the transaction layer.

8.1 TRANSPORT

The purpose of the transport layer is to accept incoming requests, process their associated security information, and parse the resulting transaction for validity via the associated schemas.

Depending on the nature of the transaction routing used, the transport layer may pass information about the context such as transaction reference numbers and authenticated sender and other third parties to the transaction routing function. Alternatively, the routing function may choose to ignore this information and rely on it being within the transaction envelope, or validate that the transport and envelope information are consistent.

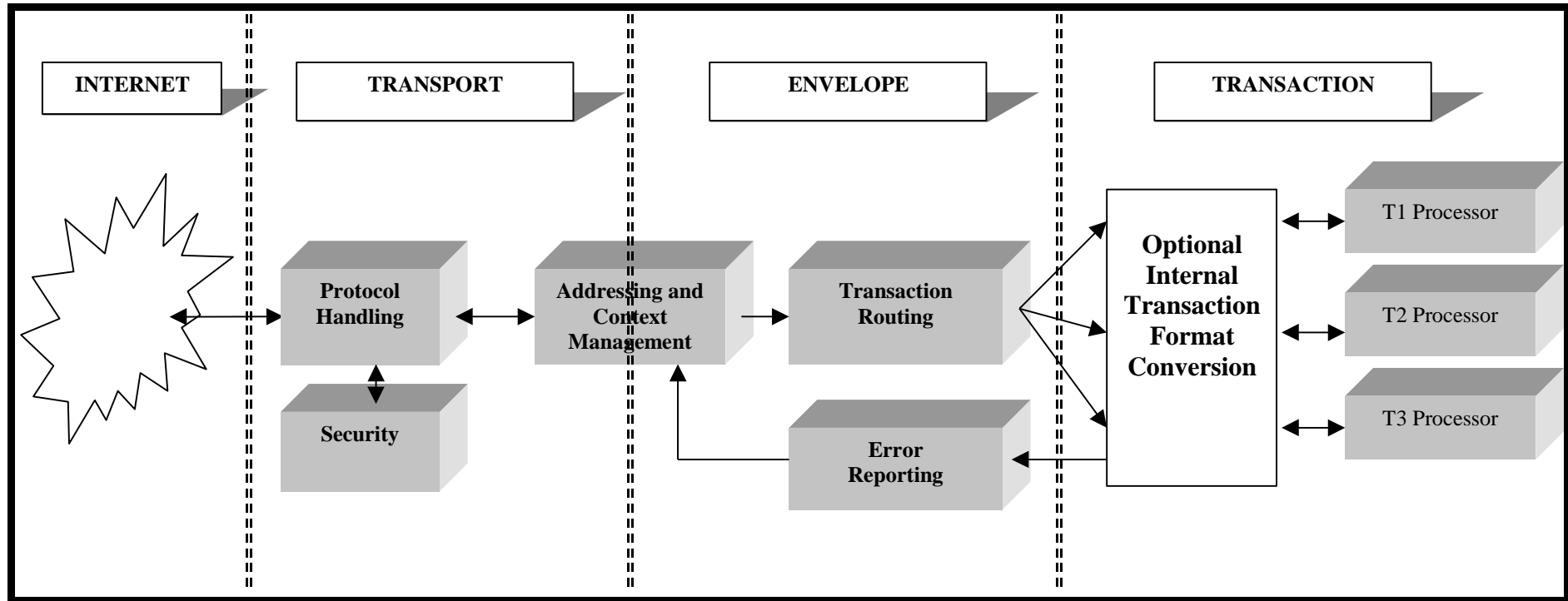


Figure 3 – High Level XML Application Architecture

8.2 ENVELOPE

The purpose of the envelope is to encapsulate all possible transactions within aseXML and provide a consistent structure for the transaction routing function to determine what transaction handler should process the transaction. In addition, it provides context from the sender that should be carried into the response to allow them to associate the response with their request.

The transaction routing function may choose to rely on the sender information provided by the transport layer, or may provide additional, application specific authentication mechanisms. The sender information may be provided explicitly (connectionless) or implied by a session handle (connection oriented) provided by the transport layer.

The envelope also provides a mechanism for consistent error reporting.

Agreement on the envelope need only be achieved once for aseXML and documented independent of individual transactions or processes.

8.3 TRANSACTION

The transaction layer is interested only in the minimal set of information necessary to process the transaction and produce the required response.

It assumes that other layers have dealt with security and access issues. The focus is on the business function rather than the IT plumbing.

9. ENVELOPE

9.1 INTRODUCTION

Having discussed the separation of envelope from transaction in chapter 8, this chapter documents the envelope to be used for aseXML.

The envelope consists of three parts:

- 1. a top level <aseXML> element**
- 2. a <Header> sub-element**
- 3. a payload sub-element.**

For transactions, the payload sub-element used is <Transactions>. For the <Acknowledgements> payload sub-element, refer to section 10.5.

The fields of the <Header> and <Transactions> sub-elements are described in detail in subsequent sections.

The entire XML tree starting with the <aseXML> element constitutes an aseXML message (see section 1.7).

9.2 <Header> SUB-ELEMENT

The purpose of the header element is to

- Identify the business parties involved in the transaction exchange.
- Uniquely identify each aseXML message.
- Provide information to allow the routing of the payload element to the appropriate application.

An example of a <Header> sub-element is shown below.

```
<Header>  
<From>...</From>  
<To>...</To>  
<MessageID>...</MessageID>  
<MessageDate>...</MessageDate>  
<TransactionGroup>...</TransactionGroup>  
<Priority>...</Priority>  
<SecurityContext>...</SecurityContext>  
<Market>NEM</Market>  
</Header>
```

Each sub-element of the <Header> is described below.

9.2.1 <From>, <To> (Mandatory)

The <From> and <To> elements identify the business parties involved.

The value of the element is the string used to uniquely identify each party.

A context attribute defines the format of the identifier. By default, National Electricity Market participant identifiers are assumed (context="NEM"), however Australian Business Numbers are also supported (context="ABN").

9.2.2 <MessageID> (Mandatory)

The sender of an aseXML message assigns it a unique identifier and places it in this element. The sender is at liberty to design the format, but it should consist only of alphanumeric characters and the hyphen character.

It is recommended that Universally Unique Identifiers (see section 1.5 reference 3) be used for MessageIDs where no alternate system exists.

It should be noted that while Universally Unique Identifiers guarantee global uniqueness, the MessageID does not need to be globally unique – it need only be unique to the sender.

This field is important in the consideration of the message acknowledgement process (see chapter 10).

Where a message is rejected (see chapter 10), a new MessageID should be allocated when it is resubmitted.

9.2.3 <MessageDate> (Mandatory)

The <MessageDate> element is the time at which the message was generated by the sender, and should be indicated to the millisecond. Note that this is not necessarily the same as the time it was delivered to the receiver.

9.2.4 <TransactionGroup> (Mandatory)

This element carries the transaction group of all the contained transactions or transaction acknowledgements.

The target application is at liberty to reject any transactions within the message that do not belong to the stated TransactionGroup.

9.2.5 <Priority> (Optional)

This element allows the sender to indicate their preference in terms of timeliness of processing for the payload. The three allowable values are

“High”, “Medium” and “Low”. It is left to the discretion of the receiver to determine whether and how to honour the requested priority.

9.2.6 <SecurityContext> (Optional)

This optional element allows the sender to provide information needed by the receiver to determine whether or not the sender is authorised to submit the transactions within the message.

For the Market Settlement And Transfer System (MSATS), this will be used to hold the participant userid from which the context for transaction processing is determined.

9.2.7 <Market> (Optional)

This optional element identifies the energy market to which the transactions in the message belong.

When not provided, a default value of “NEM” will be assumed, indicating the National Electricity Market.

9.3 <Transactions> SUB-ELEMENT

The purpose of this sub-element is to provide a container for one or more aseXML transactions. An example is shown below.

```
<Transactions>
  <Transaction transactionID="..." transactionDate="..."
    initiatingTransactionID="..." >
    <NMIDiscoveryResponse version="r100">
      ...
    </NMIDiscoveryResponse>
  </Transaction>
</Transactions>
```

Each transaction is contained within a <Transaction> element. This element carries three attributes.

9.3.1 transactionID (Mandatory)

The generator of each transaction must generate a unique identifier for it, following the same format rules as the MessageID. There need be no correlation between MessageIDs and transactionIDs generated by the same party.

This field is important when correlating response transactions to the equivalent requests (see section 9.3.3)

It is recommended that Universally Unique Identifiers (see section 1.5 reference 3) be used for `transactionIDs` where no alternate system exists.

It should be noted that while Universally Unique Identifiers guarantee global uniqueness, the `transactionID` does not need to be globally unique – it need only be unique to the sender.

Where a transaction is rejected (see chapter 10), a new `transactionID` should be allocated when it is resubmitted.

9.3.2 `transactionDate` (Mandatory)

In a similar vein to the `transactionID`, the `transactionDate` follows the same format as the `MessageDate`, and is the time at which the transaction was generated.

9.3.3 `initiatingTransactionID` (Optional)

Where the transaction is a response to a previous request, the `<Transaction>` element must also carry an `initiatingTransactionID` attribute, whose value matches that of the `transactionID` attribute of the initiating request transaction. The sender of the request is able to use this attribute to correlate responses with requests.

The specific aseXML transaction is then carried within the `<Transaction>` element. As discussed in section 3.2.9, every aseXML transaction will carry a `version` attribute.

9.4 FUTURE ENVELOPE MODIFICATIONS

It is accepted that the aseXML envelope falls far short of other frameworks currently under development in the international sphere.

In order for initial implementations of aseXML to proceed, however, a minimum set of functionality is needed to enable participants to rapidly develop their infrastructure in time for full retail competition.

The final envelope adopted will be dependent to some extent on the transport framework adopted, and the semantics it provides for recipient information and transaction context.

9.5 A SAMPLE aseXML MESSAGE

Putting together all the information presented thus far in the document, an example of an aseXML message is given below.

```
<ase:aseXML xmlns:ase="urn:aseXML:r100"
  schemaLocation="urn:aseXML:r100
  http://www.nemmco.com.au/aseXML/schemas/r100/aseXML_r100.xsd">
<Header>
  <From context="NEM">PARTICIPANT</From>
  <To context="NEM">NEMMCO</To>
  <MessageID>1324-52165-123ew</MessageID>
  <MessageDate>2000-10-31T13:20:01.000+10:00</MessageDate>
  <TransactionGroup>NMID</TransactionGroup>
  <Priority>High</Priority>
  <SecurityContext>zz023</SecurityContext>
  <Market>NEM</Market>
</Header>
<Transactions>
  <Transaction transactionID="453-333-23-WED"
    transactionDate="2000-10-31T13:20:00.900+10:00"
    initiatingTransactionID="XXX-45-WSHTY-567" >
    <NMIDiscoveryResponse version="r100">
      ...
    </NMIDiscoveryResponse>
  </Transaction>
</Transactions>
</aseXML>
```

10. ACKNOWLEDGEMENT MODEL

10.1 INTRODUCTION

The main purpose of aseXML is to facilitate transaction exchanges. It is by these exchanges that useful business is conducted. However, in order that these exchanges can occur in an orderly and traceable way, aseXML also provides a standard acknowledgement model.

The basic design philosophy for the aseXML acknowledgement model is to provide the sender with a positive acknowledgement for each aseXML message, and for each transaction within the message.

With each acknowledgement, the receiver should provide the sender with a unique identifier, called a `receiptID`, by which any queries with regard to message or transaction processing may be resolved. Whilst not currently specified, the `receiptID` would form the basis for the ability to electronically query the progress of a message or transaction.

It is recommended that Universally Unique Identifiers (see section 1.5 reference 3) be used for `receiptIDs` where no alternate system exists.

The `receiptID` is not required in the case where the message or transaction is rejected.

10.2 TRANSACTION EXCHANGES VS TRANSACTION ACKNOWLEDGEMENTS

Transaction acknowledgements can carry `<Event>` elements and hence the designer of a transaction exchange is free to use them as part of the information exchange between applications. In one sense, transaction acknowledgements are part of every transaction exchange (50% to be exact!). However, the aim of aseXML is to allow the transaction exchange designers to concentrate on the application functionality, without having to “invent” their own acknowledgment model. Hence acknowledgements are not considered part of a transaction exchange.

Put another way, where the response to a request is logically accept/reject, the designer need only define the request transaction and rely on the transaction acknowledgement to carry the response. Alternatively, where response data is required that cannot reasonably map to `<Event>` elements, or where multiple levels of acknowledgment are required, the designer will need to define their own response transaction(s).

10.3 MESSAGE ACKNOWLEDGEMENT

There may be considerable delay between the delivery of a message to the aseXML gateway and the processing of the transactions within it by

application systems. The delay is typically a result of process scheduling decisions by the receiver.

In order that the sender receive timely acknowledgement of message delivery, the receiver should respond immediately to each aseXML message with a message acknowledgement.

An example of a message acknowledgement is given below, with each attribute described in subsequent sections.

```
<MessageAcknowledgement
    initiatingMessageID="..."
    receiptID="..."
    receiptDate="..."
    status="Accept"
    duplicate="No"/>
```

10.3.1 initiatingMessageID (Mandatory)

The value of this attribute corresponds to the value of the `<MessageID>` element in the header of the message being acknowledged.

10.3.2 receiptID (Optional)

The `receiptID` is a unique identifier, assigned by the receiver of a message, to identify the processing they intend to perform as a result of receiving it. It does not need to be provided in the case where the message is rejected (see section 10.3.4).

10.3.3 receiptDate (Mandatory)

This attribute indicates the date and time at which the message was queued for processing. If the message is rejected, it indicates the date and time at which the rejection occurred.

10.3.4 status (Mandatory)

There are two possible values for this attribute, "Accept" or "Reject".

"Accept" indicates the message is accepted with no fatal errors detected.

"Reject" indicates the message was rejected. The receiver will perform no further processing on the contained transactions. The acknowledgement should carry at least one event with a severity of "Fatal".

In the case of "Reject", the message acknowledgement will contain one or more `<Event>` elements (see chapter 11) detailing the errors detected in the message. Examples might include schema validation errors.

10.3.5 duplicate (Optional)

There are two possible values for this attribute, “Yes” or “No”, the default being “No”.

When this attribute is set to “Yes”, this indicates that the receiver believes it has already processed the message and returned an acknowledgement.

It is not an error to receive a previously unseen acknowledgement that has this attribute set to “Yes”. The acknowledgment receiver may ignore the attribute. It is provided largely for logging and fault finding.

See section 10.6 for more details regarding handling duplicate messages.

10.4 TRANSACTION ACKNOWLEDGEMENT

For every transaction, a transaction acknowledgement must be sent to the originator.

The purpose of the acknowledgement is to provide the originator with an indication of the necessary information to track the progress of the request.

An example of a transaction acknowledgement is given below, with each attribute described in subsequent sections.

```
<TransactionAcknowledgement
    initiatingTransactionID="..."
    receiptID="..."
    receiptDate="..."
    status="Partial"
    duplicate="No"
    acceptedCount="20" />
```

10.4.1 initiatingTransactionID (Mandatory)

The value of this attribute corresponds to the value of the `transactionID` attribute on the container element for the transaction.

10.4.2 receiptID (Optional)

The `receiptID` is an identifier, assigned by the receiver of a transaction, to identify the processing they intend to perform as a result of receiving it. It does not need to be provided in the case where the transaction is rejected.

10.4.3 receiptDate (Mandatory)

This attribute indicates the date and time at which the transaction was queued for processing. If the transaction is rejected, it indicates the date and time at which the rejection occurred.

10.4.4 status (Mandatory)

There are three possible values for this attribute, "Accept", "Partial" or "Reject".

"Accept" indicates the transaction is accepted with no errors detected. The acknowledgement may carry "Information" or "Warning" events.

"Partial" indicates that the transaction will be processed but portions of it were in error and will be ignored. An example of this might be meter data records. The acknowledgement may carry events with any severity level except "Fatal".

"Reject" indicates the transaction was rejected. The receiver will perform no further processing of the transaction. In the case of a request transaction, no response transactions, where normally expected, will be generated. The acknowledgement should carry at least one event with a severity of "Fatal".

In the case of "Partial" and "Reject", the transaction acknowledgement will contain one or more <Event> elements (see chapter 11) detailing the errors detected in the message. Examples would include missing data or invalid data.

Where the transaction is not supported, a status of "Reject" will be used, with the <Event> element indicating this error condition.

Where the receiver does not support the version of the transaction, a status of "Reject" will be used, with the <Event> element indicating the versions of the transaction supported by the receiver.

10.4.5 duplicate (Optional)

There are two possible values for this attribute, "Yes" or "No", the default being "No".

When this attribute is set to "Yes", this indicates that the receiver believes it has already processed the transaction and returned an acknowledgement. It is not an error to receive a previously unseen acknowledgement that has this attribute set to "Yes". The acknowledgment receiver may ignore the attribute. It is provided largely for logging and faultfinding.

See section 10.6 for more details regarding handling of duplicate transactions.

10.4.6 acceptedCount (Optional)

Where the transaction contains multiple entries that are processed simultaneously, this attribute may be used to indicate the number of entries that were accepted. Typically, events will be provided to indicate any entries that were not accepted.

The major use of this attribute is where the transaction carries CSV format data (see chapter 13).

10.5 EXCHANGING ACKNOWLEDGEMENTS

All message and transaction acknowledgments will be carried in an aseXML message within a payload element of <Acknowledgements>.

Messages with an <Acknowledgements> payload containing message acknowledgments will not themselves be acknowledged.

Multiple acknowledgements of both types may be carried in a single payload, with those for messages preceding those for transactions.

Note that where both message and transaction acknowledgements are carried together, the previous paragraphs imply that no corresponding message acknowledgement will be generated. If tracking of the delivery of transaction acknowledgments is considered important, they should be transferred using separate acknowledgement messages.

Where transaction acknowledgements are carried, they will all correspond to transactions of the same <TransactionGroup>. The TransactionGroup value will be included in the header, consistent with its use for the corresponding transactions.

Where only message acknowledgements are carried, a <TransactionGroup> of "MSGs" will be used.

The aseXML acknowledgement model allows for multiple messages to be acknowledged via a single acknowledgment message. Similarly, transactions from multiple messages could be acknowledged together, provided the rules for TransactionGroup of the acknowledgement message are not violated.

Whilst grouping acknowledgements may lead to better use of transport bandwidth, it is a matter for the binding to a particular transport to decide whether this is permitted.

10.6 HANDLING DUPLICATES

If a sender believes that a message or transaction was not received by the intended target, they may resend the message or transaction. This can result in duplicate messages/transactions being received.

The recipient of a duplicate message/transaction should carry out the following on receipt:

- if the receiver has not already generated and returned an acknowledgment, the assumption is that the original message/

transaction has not yet completed internal processing. In this instance the receiver should not process the duplicate further, as the original acknowledgment will eventually be sent to respond to the transaction. All duplicates received whilst generating the original acknowledgment should **not** in turn be acknowledged, as the original acknowledgment will complete the transaction/acknowledgment cycle.

- any duplicate transactions received after the original ack has been sent must assume the original acknowledgment was lost in transit. On receipt the recipient must send an acknowledgment with the same receiptID and status as the original acknowledgement, but with the duplicate attribute set to "Yes". The acknowledgement date should reflect the date/time at which the second acknowledgement was generated.

10.7 A SAMPLE aseXML TRANSACTION EXCHANGE

The diagram below provides an example of a transaction exchange between a (S)ender and a (R)eceiver. Each line represents an aseXML message, with some elements and attributes omitted for clarity.

H() indicates the contents of the message header, whilst T() and A() represent the <Transactions> and <Acknowledgements> payload elements.

The diagram shows the sender generating a message containing three transactions (1). The message (2) and then the transactions (3) are acknowledged by the receiver. The sender acknowledges the transaction acknowledgment message (4). The receiver then generates a response transaction (5) to the first of the three in the initial message. This response message (6), then the transaction (7) it contains are acknowledged. The receiver then acknowledges the transaction acknowledgment message (8).

“m=” refers to a <MessageID> element value.

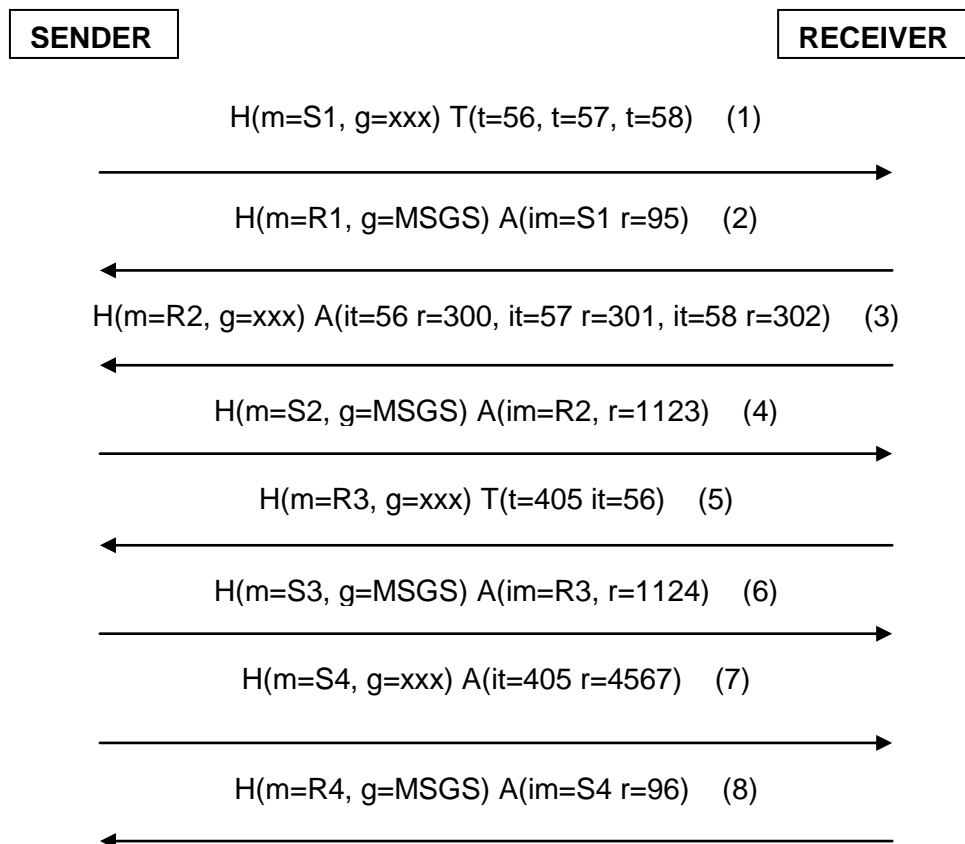
“im=” refers to a initiatingMessageID attribute value.

“t=” refers to a transactionID attribute value.

“it=” refers to a initiatingTransactionID attribute value.

“r=” refers to a receiptID attribute value.

“g=” refers to a <TransactionGroup> element value.



11. ERROR REPORTING AND THE <Event> ELEMENT

Error reporting is an important function of message and transaction acknowledgements. Errors will also need to be reported in response transactions. In order for errors to be reported consistently, aseXML defines a standard <Event> element for this purpose.

Zero, one or more <Event> elements are supported within a <MessageAcknowledgement> or a <TransactionAcknowledgement> element.

It is up to the designer of a transaction exchange to decide how to incorporate application error reporting. In general, a response transaction should support content incorporating the normal response and <Event> elements.

The example event element below indicates that a schema error has occurred. Subsequent sections describe the attributes and elements of the <Event> element.

```
<Event class="Message" severity="Fatal">
  <Code>2</Code>
  <KeyInfo>Line number or other info</KeyInfo>
  <Context>The contents around the error</Context>
  <Explanation>Further text describing the error</Explanation>
</Event>
```

11.1 class ATTRIBUTE (Optional)

All events fall into one of the following classes.

- Message

The message class covers validation of the aseXML message structure. Examples of errors at this level include inconsistent header elements, unsupported transactions and unsupported transaction versions.

- Application (default)

This class covers application level validation. Events of this class will normally only appear in <TransactionAcknowledgement> elements or in response transactions.

- Processing

The processing class covers environmental issues. An example might be the long-term unavailability of target applications or the corruption of a database.

11.2 severity ATTRIBUTE (Optional)

The severity attribute indicates the nature of the event, and takes one of the following values.

- Information

Processing is unaffected by the contents of the event. It is provided for general interest. In the absence of any other circumstances, this severity attribute should be used for a code of 0 (see section 11.8).

- Warning

Processing may proceed by application of overriding processing rules. An example might be substitution of a default value for a missing optional element.

- Error

An error is present that must be corrected. Processing may still continue. An example might be an invalid meter data record that is unrelated to the remainder of the records presented for processing.

- Fatal (default)

The nature of the error is such that further processing is not possible.

11.3 <Code> SUB-ELEMENT (Mandatory)

This element is a numeric code corresponding to the particular event condition. Values from 0 to 999 are reserved for definition by the aseXML standard. The intention is to provide a common set of values covering most situations, allowing consistent interpretation of codes. The currently defined list is shown in section 11.8. Where the code is not in the reserved range, a `description` attribute should also be provided according to the guidelines in section 2.5.

The following table defines the ranges of codes currently allocated within aseXML.

Use	Range
aseXML Reserved	0-999
National Electricity Market	1000-1999
New South Wales Gas Market	2000-2999
Victorian Gas Market	3000-3999
Faults	4000-4099

11.4 <KeyInfo> SUB-ELEMENT (Optional)

Where the combination of class and code are insufficient to completely describe an event, this element may be used to provide further detail as to the information needed to locate the source of the event within the original transaction.

For CSV data carried as the content of an element, the value of the <KeyInfo> field should be the key column values for the line in error, separated by commas if necessary.

11.5 <Context> SUB-ELEMENT (Optional)

This element should contain the portion of the input to which the event applies.

11.6 <Explanation> SUB-ELEMENT (Optional)

Where the code used is of a generic nature, and further explanation is required, this information should be provided via this element.

Another example of an event is provided below, in this case of an event generated for an unknown transaction version.

```
<Event class="Message" severity="Fatal">
  <Code>4</Code>
  <SupportedVersions>
    <Version>r90</Version>
    <Version>r95</Version>
  </SupportedVersions>
</Event>
```

11.7 <SupportedVersions> SUB-ELEMENT (Optional)

Where the condition of an unsupported transaction version is indicated, the event should include the <SupportedVersions> element. It indicates the versions of the transaction that are supported by the receiver via one or more <Version> sub-elements.

11.8 Reserved Event Codes

Class	Code	Description	Notes
	0	Success, OK, Accepted, etc.	Any class
Message (1-99)	1	Not well formed	
	2	Schema validation failure	
	3	Transaction not supported within Transaction Group	The transaction is not supported by the receiving system in the context of the provided transaction group
	4	Transaction version not supported	
	5	Uncompression failure	This covers both errors in the uncompression process and the absence of the appropriate file within the compressed format container
	6	Message too big	
	7	Header mismatch	Information provided by transport layer is inconsistent with the message header
	8	Incorrect market	The system to which the message is addressed does not handle the market indicated in the header
	9	Unknown Transaction Group	The transaction group is not supported by the receiving system

Class	Code	Description	Notes
Processing (100-199)	100	Application unavailable	
	101	Database data error	Typically the result of code error, such as insufficient checking of data validity prior to insertion into the database.
	102	Database system error	e.g. major database problem
Application (200-998)	200	Record(s) not found	
	201	Data missing	
	202	Data invalid	
	203	Unknown report	Requested report not supported by receiving system
	204	Missing or invalid report parameters	
	205	Unknown Table	Requested table is not replicated by the receiving system
	206	Unknown initiating Transaction ID	We didn't send this transaction – no record of initiating transaction ID. Therefore the response transaction does not belong to us.

Class	Code	Description	Notes
Other	999	Unexpected Error	Any Class Must be accompanied by an <Explanation> element Only to be used where an error can't reasonably be mapped to an existing error code optionally carrying an <Explanation> element to further explain the specific nature of the error

12. GENERIC TRANSACTION EXCHANGES

Some transaction exchanges lend themselves to be used within more than one TransactionGroup, for instance reports. aseXML allows a transaction exchange to be supported within more than one TransactionGroup (see section 1.7).

Depending on the way these “generic” transactions are designed, they can be extended to accommodate use in new TransactionGroups without affecting their basic structure. Typically this involves designing the transaction on the basis of one or more abstract types, and developing derived types specific to each TransactionGroup that wishes to use the transaction.

12.1 TABLE REPLICATION

Section 2.5 allows for descriptions to be omitted on codes in the event that “mechanisms are in place for the exchange between businesses of the code/description mapping information”. Table replication provides such a “mechanism”. Whilst initially designed for low volume applications such as codes, it is sufficiently generic to allow replication of arbitrary amounts of any information that can be expressed as a table.

The “table” paradigm is borrowed from the relational world, and represents data as a series of fixed format rows within a table. The table should be considered as a logical entity, and need not have a physical representation (though it more than likely will) within the providing system.

Once created, a row can only be updated once more and then only to indicate that it has been superseded by another row. A logical update is achieved by creating a new row with identical data except for those columns that have changed.

All rows thus carry a `CreationDate` indicating the date/time at which they were created, and a `MaintenanceDate` indicating the date/time at which they were superseded. In addition, the status field initially starts with a value of “A” for active, and is replaced with an “I” for Inactive when the row is superseded.

A system providing table data may place a limit on the number of rows returned by any one request. In order that the remaining rows can be retrieved, every row carries a non-zero integer sequence number. A sequence number is provided with the request, meaning that only rows with a greater sequence number should be returned. Returned data should also be sorted by sequence number. As a result, a table can be provided in “chunks” by providing a sequence number of zero on the initial request, and repeating the request with the maximum sequence number from the output of the previous request, until such time as no further rows are returned.

The table replication request transaction allows the provision of a table name, date range and sequence number. The date range selects only those rows whose creation or maintenance date falls inclusively within the specified range. A low date of 2001-01-01 and a high date of 9999-12-31 should be used where the date range is not relevant.

The table replication notification transaction is based on an abstract table type. Specific types for tables within a TransactionGroup are then derived from this base type. Note that these specific types may be used across TransactionGroups as appropriate.

The term notification rather than response is used because systems may choose to notify others of incremental changes to tables asynchronously. Whilst only one table at a time may be selected in a replication request, the notification transaction allows the inclusion of data from multiple tables, each being held within a ReplicationBlock. Instances indicate via an `xsi:type` attribute the specific table types being provided.

An example of a replication request transaction is shown below.

```
<ReplicationRequest version="r5">
  <ReplicationParameters>
    <TableName>DistributionLossFactorCodes</TableName>
    <CreationFromDate>2000-01-
01T00:00:00.000+10:00</CreationFromDate>
    <CreationToDate>9999-12-
31T00:00:00.000+10:00</CreationToDate>
    <LastSequenceNumber>0</LastSequenceNumber>
  </ReplicationParameters>
</ReplicationRequest>
```

12.2 REPORTS

The report transaction exchange uses two abstract types, one for report parameters and one for the format of the resulting report. Instances indicate via an `xsi:type` attribute the specific type used for the provision of parameters and the resulting output format.

As a minimum, all report parameter types must include the name of the report required. Note that multiple reports may use the same report parameter type in the event that the input parameters are identical. In this case, only the report name will vary.

A copy of the parameters is provided with the resulting report.

Two standard report format types are provided for CSV style output, or output in the same format as used in the ReplicationNotification transaction.

An example of a report request transaction is shown below.

```
<ReportRequest version="r5">
  <ReportParameters
    xsi:type="ase:CATSStatisticsReportParameters">
    <ReportName>Statistics</ReportName>
    <FromDate>2000-01-01</FromDate>
    <ToDate>9999-01-01</ToDate>
    <Public>Yes</Public>
  </ReportParameters>
</ReportRequest>
```

13. SUPPORT FOR CSV FORMAT DATA

For high volume, repetitive data, it may be considered appropriate for this to be carried in CSV format within a transaction element.

13.1 FORMAT CONSTRAINTS

Data within a given column of the CSV data should have the same meaning for ALL lines. CSV format data utilising a pseudo-tagged structure, whereby particular lines or columns are used to interpret the meaning of other lines or columns, is NOT supported by aseXML.

The first line of any CSV data should consist of column designators. The purpose of the designators is twofold;

- Column interpretation is able to be position independent.

Applications processing the CSV data must utilise the designators to determine the column meaning, and should NOT assume the columns will always be delivered in a fixed order.
- Products such as Microsoft Excel or Oracle SQL*Loader can utilise the column designators to more usefully process the subsequent data lines.

13.2 LINE TERMINATOR

The ASCII carriage return character should be used to indicate the end of each line of CSV data. The last line may be terminated by the enclosing tag and need not end with a carriage return.

To allow for flexibility in terms of the tools used to manipulate CSV data, any line feed characters present at the start of a line of CSV data should be removed prior to processing of the line.

Thus a line ending in a carriage return/line feed pair would result in the line being terminated by the carriage return, the line feed appearing at the start of the next line, and this line feed character then being stripped from the subsequent line prior to it being processed.

14. ACCESSING aseXML SCHEMAS AND INSTANCE DOCUMENT EXAMPLES

The schemas and example messages/transactions for each version of aseXML are accessible via www.aseXML.com.

15. MESSAGE SERVICES

Messaging services used to deliver aseXML instance documents are discussed in this section.

15.1 AseXML FTP hokey-pokey protocol

This protocol is detailed in the section “8.3 Managing Files” in the document “010905 - Technical Architecture Design Report v4.4.pdf”.

15.2 Using aseXML with ebXML messaging service

The Victorian Gas market uses ebXML Message Service binding with HTTP to deliver aseXML message as attachments as defined in the “FRC B2B System Specifications v1-6.doc”.

15.3 AseXML binding with SMTP

This is a secondary means for delivering aseXML messages over SMTP, i.e. via the email. As defined in the “SMTP Binding to aseXML Specification v0.7.doc”.