NEMMCO

National Electricity Market Management Company Limited
ABN 94 072 010 327

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

Prepared by: Michael Leditschke

Version No: 1.2

DRAFT

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.nemmco.com.au/aseXML)
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

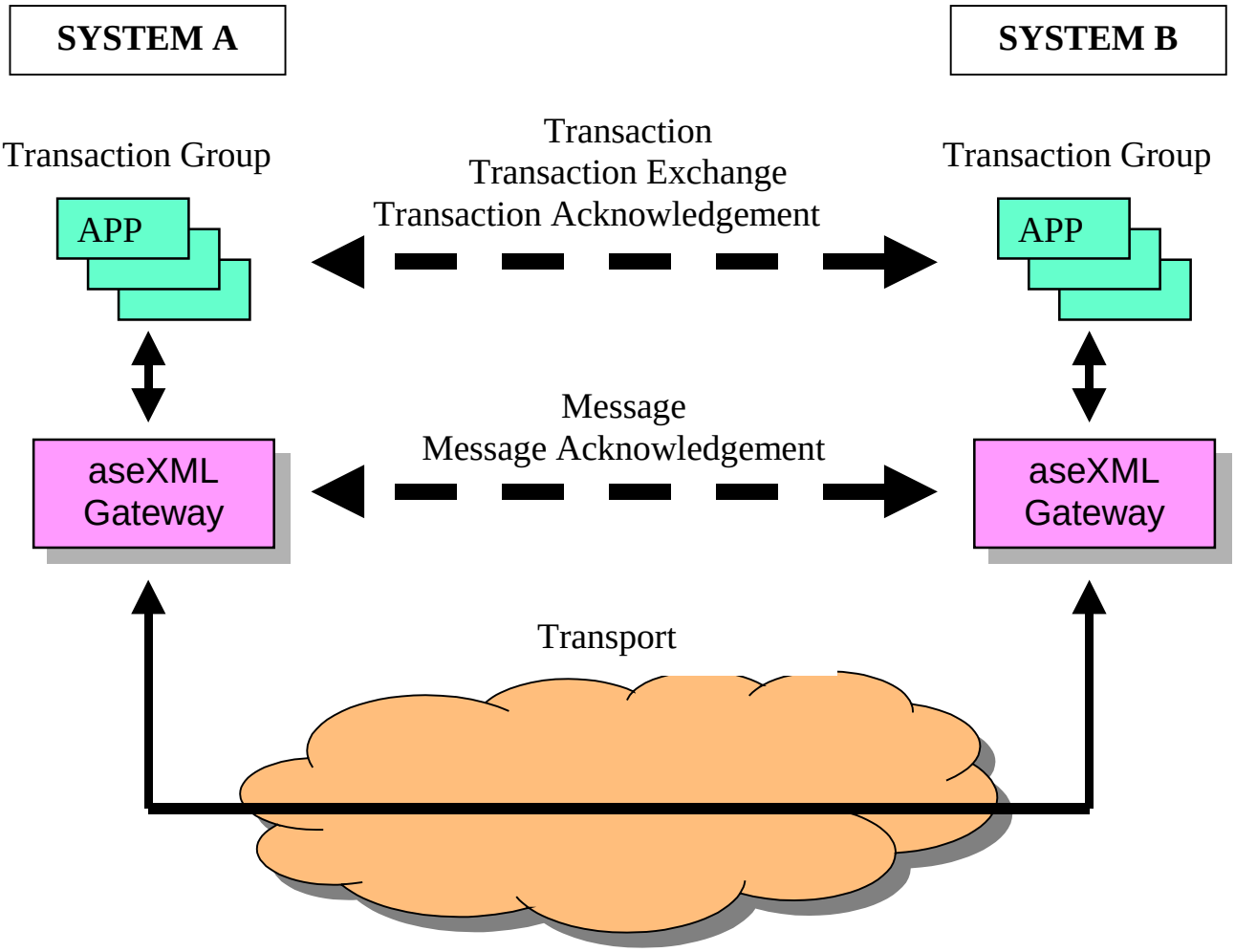
Words such as "transaction", "message", "acknowledgement" and "gateway" are commonly used in a wide variety of contexts within the Information Technology Industry.

It is thus important to understand their use within aseXML. The diagram 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.



aseXML Conceptual Model

In order to prevent circular acknowledgements, there is no acknowledgement of transaction acknowledgements.

A **transaction group** identifies a set of related transaction exchanges. Each transaction exchange is associated with one or more transaction group.

Transaction groups are intended to assist an aseXML Gateway (see below) in prioritising and routing transactions to the appropriate application within an end system. Thus from an aseXML perspective, a transaction group identifies an “application” within an end system.

An aseXML **message** provides a standard envelope for the carriage of transactions or acknowledgements. One message can carry multiple transactions or acknowledgements. Within a given message, all transactions or transaction acknowledgements must relate to the same transaction group.

For each message, the receiving gateway generates a **message acknowledgement**. A message acknowledgement allows tracking of the message’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 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

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

1.8 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 0.5	09/10/2000 31/10/2000	Michael Leditschke Michael Leditschke	Add additional element naming guidelines 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 24th 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

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

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

Version	Date	Who	Comments
			Added `duplicate` attribute definition and description to acknowledgements Added `acceptedCount` attribute definition and description to transaction acknowledgement
1.2	27/06/01	Michael Leditschke	Introduced the concept of generic transaction exchanges that can appear within multiple `TransactionGroups`, e.g. reports and table replication Relaxed the restriction in section 1.7 that a transaction exchange may only appear in one TransactionGroup to allow for generic transactions Added chapter 12 on generic transaction exchanges Added additional standard event codes in section 11.8 Completed hanging sentence in section 10.2 (thanks James)

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 assuring 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.

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.

3. VERSION CONTROL

3.1 XML AND VERSIONING

3.1.1 Guiding Principles

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.

In selecting a versioning approach for aseXML, the author has attempted to pick the “middle road” that ensures possible changes in versioning strategy are not precluded, whilst not unduly complicating the generation and processing of transactions. 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.

- Minimise the amount of version information within instance documents.

  This ensures instance documents are simple to generate and read.

- 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.

- Accommodate the need for applications to make logic decisions on the basis of version.

  Any version mechanism should provide version information to applications in a manner that is simple to handle programmatically.

3.1.2 What Should Be Versioned?

Given the de-centralised, incremental design process being used for aseXML, it is vital that robust mechanisms exist to allow participants to track and implement changes to the standard. Such mechanisms should also be efficient in terms of likely application architectures.

There are three areas where changes to aseXML are likely to occur;

- in **common data items**, that, by definition, are used by multiple transactions.

Once defined, common data items are unlikely to change, with the exception of enumerations. Given the discussion in section 2.4, such changes will be infrequent.

When changes to common items do occur, however, multiple transactions will be affected. In turn, application processing may be affected, depending on how the nature of the data involved has changed and what IT architecture is involved.

For instance, addition of values to an enumeration may require code change when the code value triggers application logic, but may not require code change if the value is simply stored for later display.

As another example, an increase in the length of a data item may cause the need for a database to be resized, but may not cause a problem if mapping of the XML transaction to an internal structure occurs, and the internal structure has the capacity to handle the expansion.

- in the **transactions**, which combine common and transaction specific data items.

  Transactions are considered the area where most change will occur, given the distributed nature of their specification (see section 1.2). Changes in transactions are also seen as the most likely driver for changes to application processing.

- in the **envelope**, which ensures delivery of the transactions (see chapter 9).

  The envelope is only likely to change with the move from the interim transport solution to the final aseXML framework. This change should not affect code specific to particular transactions, but will require changes to the code used to route transactions to the appropriate application.

From the above discussion, the versioning mechanism selected must provide the necessary hooks to allow application code to detect and handle variations in the common data items, transactions and envelope in a way that does not mandate or preclude a particular application framework.

3.2 VERSIONING AND XML NAMESPACES

The “Namespaces in XML” specification provides a starting point for considering versioning issues. Quoting from the 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 the 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 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.

3.2.1 Namespace Granularity

In formulating a standard such as aseXML, one of the design decisions to be made is how many namespaces to use. 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 following table summarises the options for aseXML and their advantages/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 transaction/common element variations need to be considered
Namespace per transaction Namespace for common items	High	Fine version control	• Large number of namespaces • Complex management at application level • Use of multiple schemas complicates schema design and instance documents
Namespace per transaction set	Medium	• Parallels likely participant support of portions of the specification • Reasonable granularity	• Complex management at application level • Use of multiple schemas complicates schema design and instance documents

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

3.2.2 Namespaces Within aseXML

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.

Instance documents will qualify their top-level element with the aseXML namespace corresponding to the version of the contained transaction(s).

The reasons below were used to arrive at this decision.

- Use of one namespace is in line with the guiding principles of section 3.1.1, that is simplicity of schemas and simplicity of instance documents.
- Some schema parsers (see section 3.3) 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.

It is recognised that this approach will need alternate mechanisms to handle changes in transactions and common items in order to meet the third guiding principle of simple application version management.

3.3 VERSIONING AND 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.

3.3.1 Schemas Within aseXML

Given that different products may use different strategies to obtain schemas, it is not possible to be prescriptive in this standard. 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 all instance documents 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 3.7, 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.

3.4 RELEASE IDENTIFIERS

In order to further define the versioning scheme for aseXML, it is necessary to document how different versions of aseXML will be identified.

To this end, a release identifier will identify each version 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

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 3.7.

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

3.5 VERSIONING OF TRANSACTIONS

It is recognised that by adopting a single namespace, tracking of changes in individual transactions by application code becomes somewhat of a moving target. The current production release will most likely be different from that used to test the application code, and the ability to respond to arbitrary production releases of the standard will be difficult to support.

Within aseXML, a number of "release points" will thus be assigned to each transaction. A release point indicates the production release at which the contents of the transaction, and hence the associated application semantics, changed.

Associated with each transaction will be a `version` attribute indicating the release point to which the transaction conforms.

When an application generates an instance document containing a particular release point of a transaction, it should associate it with the namespace corresponding with the release point.

This approach has several features.

- Application code for a given transaction need only know how to generate and process a limited set of production releases. Code should be structured to use the version attribute to control variations in processing.
- Code written to generate a particular version of a transaction will not be invalidated when the version identifier of the aseXML namespace changes as a result of modifications in some other transaction.
- 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 1.4).
- Given that a query mechanism is available, an application having a given version of a transaction 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.

- Regardless of the version of the namespace, examination of its schema will quickly reveal the release point of each transaction, since the definition of the transaction will be carried forward with each production version.

Whilst at first glance appearing somewhat complicated, the approach above will allow participants to choose what subset of the release points of each transaction they implement, and does not restrict those participants who wish to aggressively advance their IT infrastructure. Figure 1 provides an example of how changes in transaction T1 are accommodated by the aseXML schemas and by participant infrastructure.

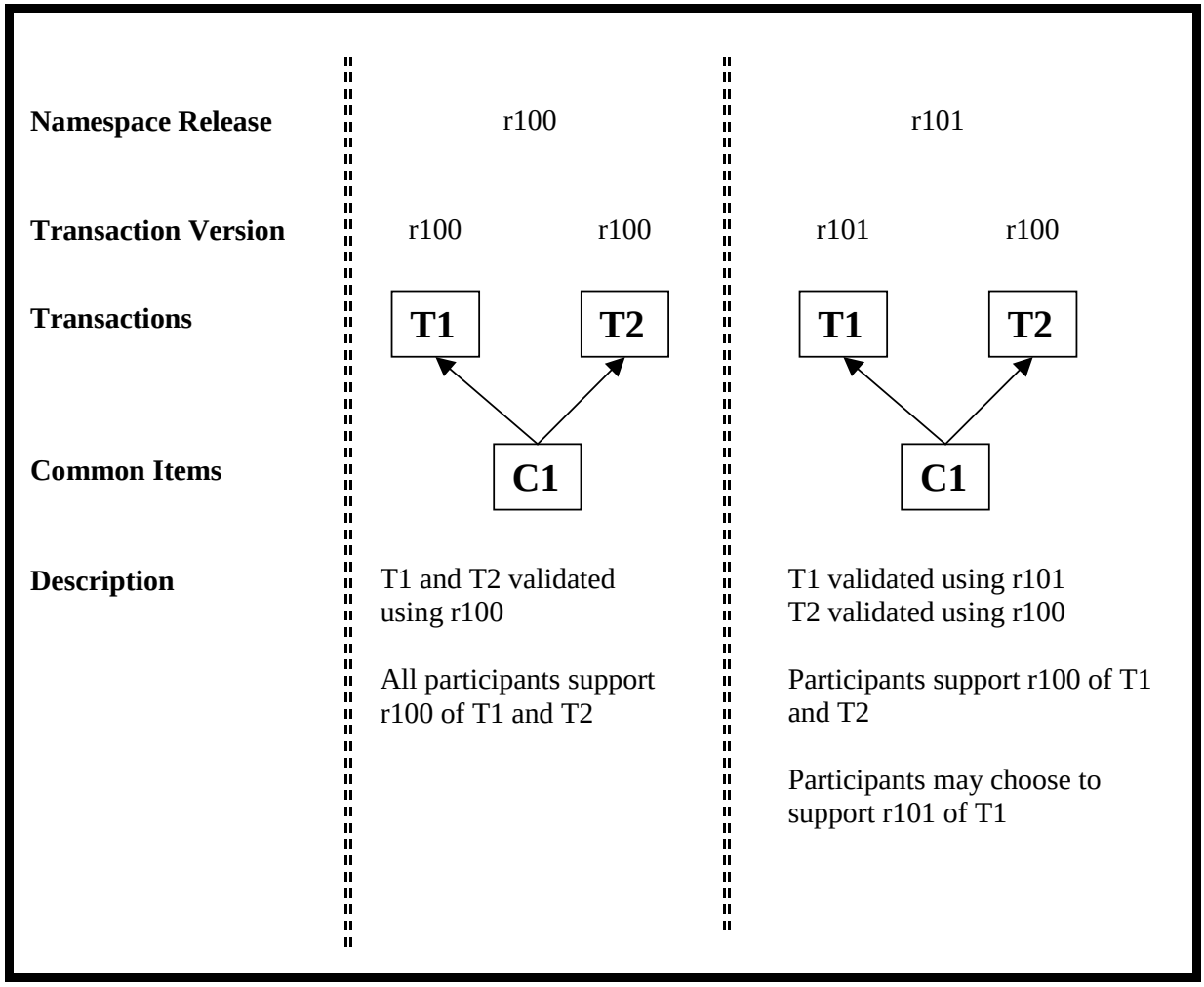


Figure 1 – Transaction Versioning in aseXML

3.6 VERSIONING AND COMMON ITEMS

Changes to common items will affect multiple transactions. Two approaches to handling this problem are available

- change the release number of all transactions using the common item.
- support multiple versions of the common item, and only update the release number of the transactions needing the updated common item.

The mechanism already outlined for transaction versioning inherently provides some support for different versions of common items, since these will reside in different namespaces. However, any subsequent changes to transactions not initially affected by changes to a common item must incorporate the newest version of the common item.

Support for multiple versions of common items within a single namespace is also possible within XML schemas. By limiting common items to type definitions, and incorporating version information in the type names, instance documents can be kept "version free" whilst the schemas provide strict type checking.

In considering the approach to adopt in aseXML, it was decided that the facilities provided by incorporating versioning in the aseXML namespace were adequate for the moment, given that changes to common items should be rare. In order to facilitate the possible use in the future of version information in type definitions, however, the following guidelines should be followed.

Common items will be limited to type definitions. No common elements will be declared.

Use of the "ref" facility of XML schemas to refer to common elements is prohibited. Note that use of the "ref" facility is not prohibited where use of global elements is considered desirable.

A change to a common type definition will only result in a release change for those transactions requiring the modified type definition. Whenever a change is made to a transaction, however, the semantics of the latest version of all common type definitions used must be incorporated with any other changes.

3.7 THE BIG PICTURE – INTRODUCING A CHANGE TO aseXML

This section presents a scenario to demonstrate the use of proposed version control mechanisms.

3.7.1 Scenario

aseXML is at release r100. It becomes evident that a new production version is needed as a result of changes to the operation of meter data access within the market.

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.

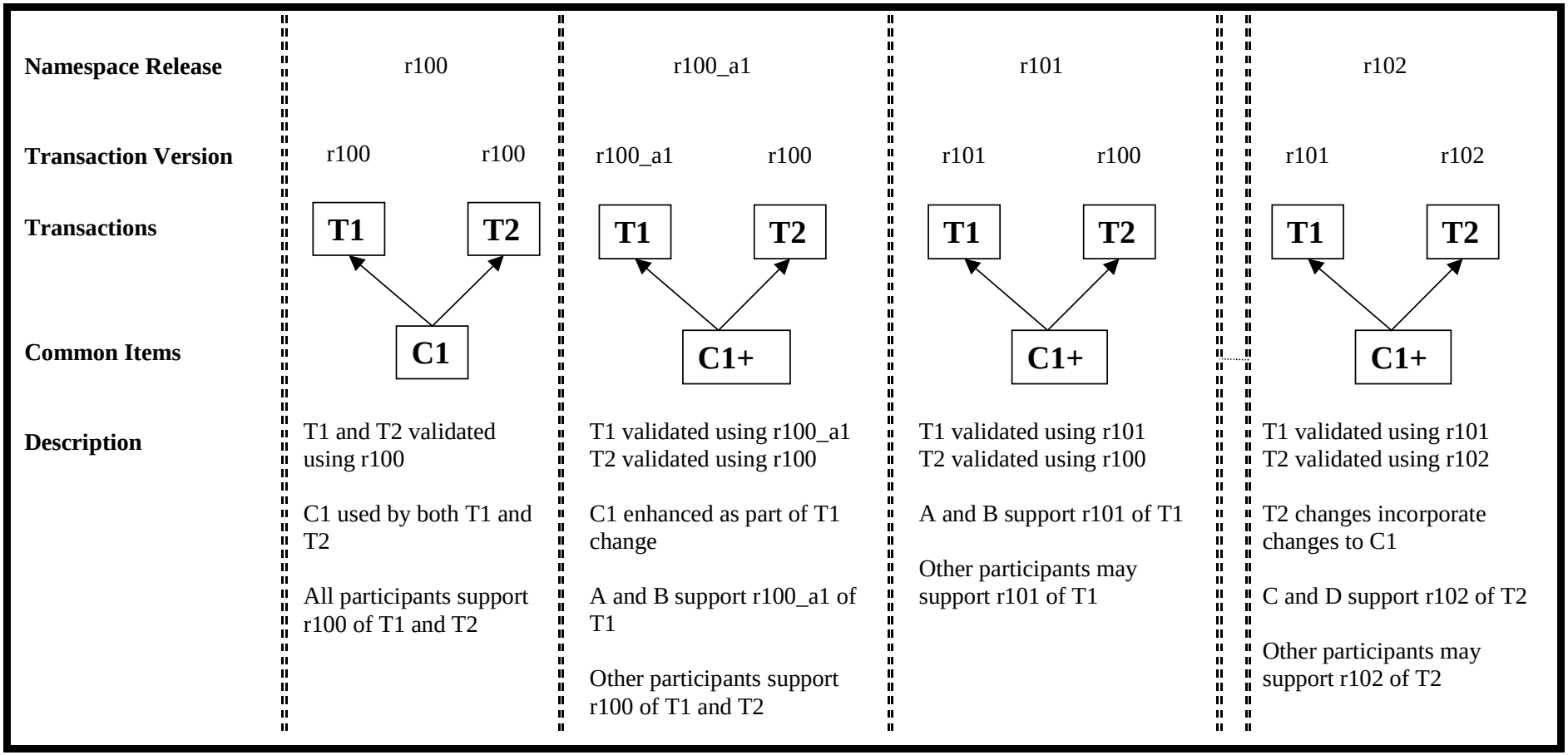


Figure 2 – Introducing A Change To aseXML

3.7.2 Sequence of Events

1. The letter “a” is assigned to the development thread.

2. A and B communicate privately and decide upon a first cut of the changes.

3. A copy of the current production version of the aseXML namespace schemas is taken.

4. The transaction being altered may refer to common types. Any application changes as a result of previous changes (from prior releases) to these types must be incorporated as part of his release.

5. A and B agree on a development extension. A and B choose r100_a1.

6. In this case, the length of a DLF Code needs to be increased by one character.

7. All references to the namespace in the schema files are updated to the development release, together with the version identifiers within the schema filenames. The version attributes of the affected transactions are updated to those of the new namespace.

8. A and B enhance their infrastructure to support the changes. There may be multiple iterations and depending on the schema infrastructure used, the development version may change as agreed by A and B.

9. A and B are ready for interoperability testing and feel the change is ready for public scrutiny.

10. An area within the web site containing the aseXML schemas is created for the development version and the complete schema is placed on the site.

11. As a result of testing between A and B and public comments, steps 5 to 10 may be repeated.

12. 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.

13. 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.

14. Agreement is reached amongst participants that the change is ready for production release.

15. The next production release is assigned and step 7 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.

16. An area within the web site containing the aseXML schemas is created for the production version and the complete schema is placed on the site. The files for all development versions of this thread are deleted from the site.

17. Other participants may now implement this release point of the transaction according to their IT schedules.

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:ReleaseIdentifier

where

- **ReleaseIdentifier is the release identifier of the namespace as per section 3.4.**

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
World Wide Web Consortium	
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**
aseXML	
urn:aseXML:r?	**ase**

5. SCHEMA ORGANISATION

5.1 SCHEMALOCATION URLs

As per the guidelines discussed in section 3.3.1, 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/ReleaseIdentifier/aseXML_ReleaseIdentifier.xsd

where

- **WebSiteRoot is the root portion of the URL needed to gain access to the web site.**
- **ReleaseIdentifier is that of the corresponding namespace and complying with section 3.4.**

Thus, an example of a URL might be

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

The ReleaseIdentifier is included in the filename portion of the URL so that it remains unique even when separated from the rest of the URL, for instance in a local parser cache. It is also included in the URL path in line with section 3.7.

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

where

- **ApplicationTitle is replaced with the short title of the application. It may contain alphanumeric characters and will use title case.**
- **ReleaseIdentifier is that of the corresponding namespace and will comply with section 3.4.**

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_ReleaseIdentifier.xsd

where

- **ExchangeTitle is replaced with the short title of the transaction exchange. It may contain alphanumeric characters and will use title case.**
- **ReleaseIdentifier is that of the corresponding namespace and will comply with section 3.4.**

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 3.4.**

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 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 PROLOG

All schemas will include an XML prolog including the version attribute.

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 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.

7. INSTANCE DOCUMENTS

7.1 XML PROLOG

All instance documents will include an xml prolog identical to that of the schemas.

7.2 DEFAULT NAMESPACES

Default namespaces will not 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.

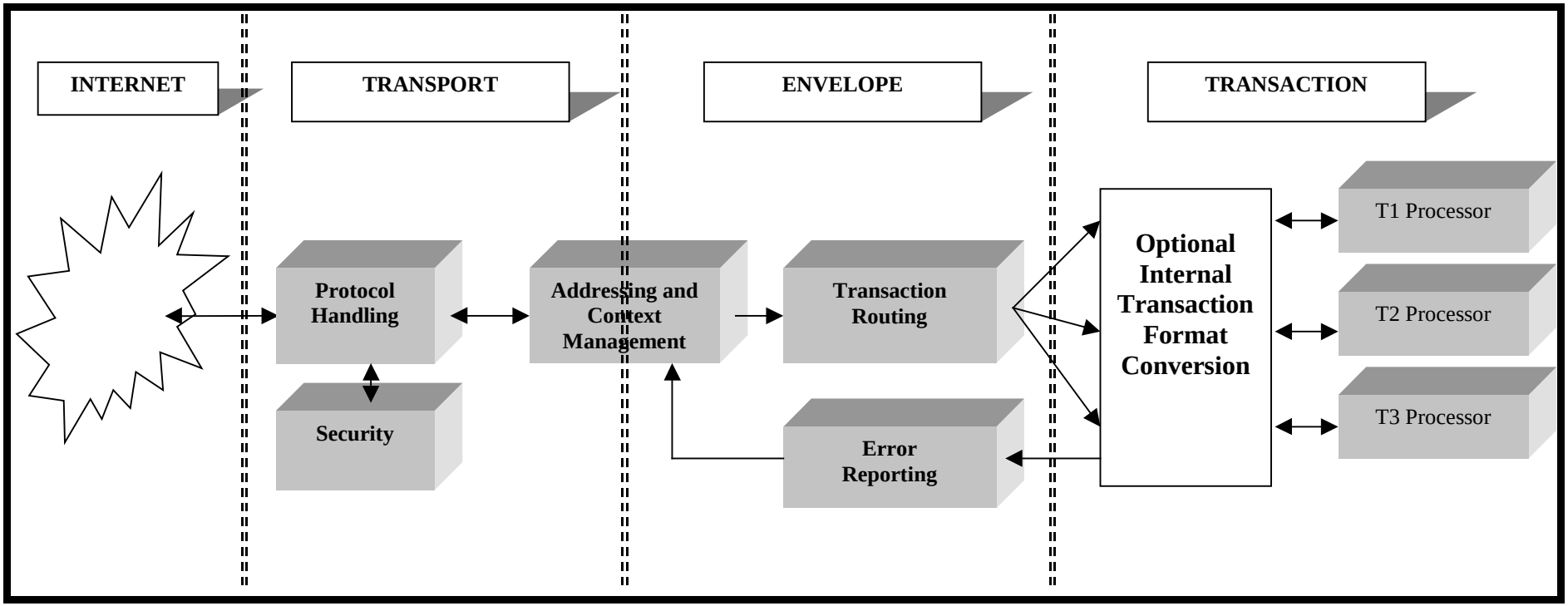


Figure 3 – High Level XML Application Architecture

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.

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.

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 3.5, 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 xmls: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 `receiptID`s 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.

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 initatingMessageID (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”.

Where the receiver believes it has already processed the message, it should return an acknowledgement with the same receiptID as the original acknowledgement, but with this attribute set to “Yes”. The acknowledgement date should reflect the date/time at which the second acknowledgement was generated.

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

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 initatingTransactionID (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 ”Informational” 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”.

Where the receiver believes it has already processed the transaction, it should return an acknowledgement with the same receiptID as the original acknowledgement, but with this attribute set to “Yes”. The acknowledgement date should reflect the date/time at which the second acknowledgement was generated.

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

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 acknowledgements 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 will its use for the corresponding transactions.

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

The aseXML acknowledgement model aseXML 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 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 receiver then generates a response transaction (4) to the first of the three in the initial message. This response message (5), then the transaction (6) it contains are acknowledged.

"m=" refers to a `<MessageID>` element value.

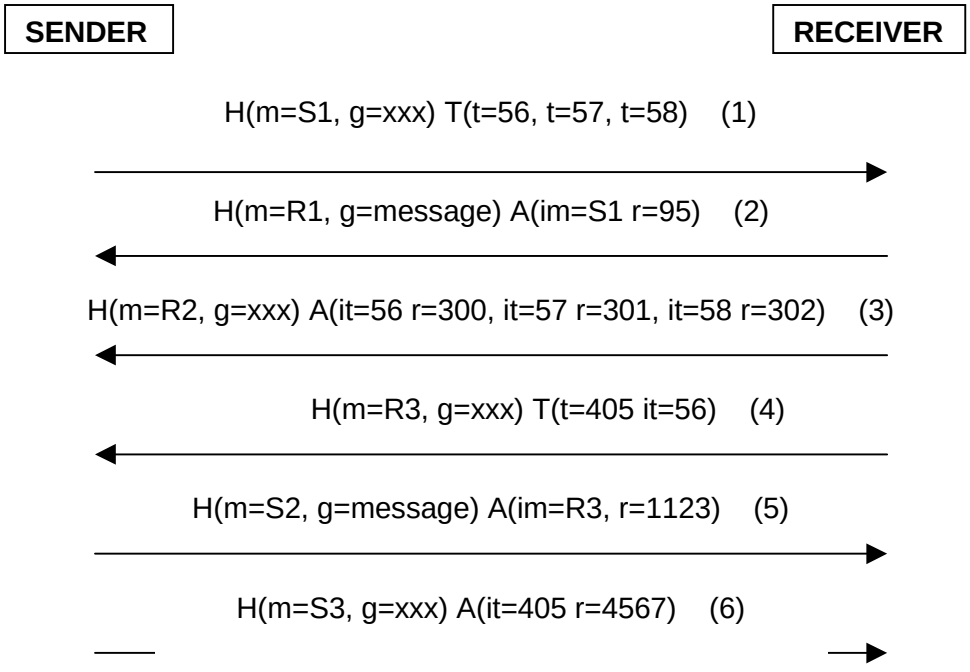
"im=" refers to a `initiatingMessageID` attribute value.

"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.

- 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 range 1000 to 1999 is reserved for use by the MSATS system.

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 uncompress ion 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

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 (200-999)	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

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 subsequently updated once, and then only to indicate that it has been superceded 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 superceded. 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 superceded.

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 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.

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;

1. 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.

2. Products such as Microsoft Excel or Oracle SQL*Loader can utilise the column designators to more usefully process the subsequent data lines.

14. ACCESSING aseXML SCHEMAS AND INSTANCE DOCUMENT EXAMPLES

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

www.nemmco.com.au/aseXML