Creating A Single Global Electronic Market

ebXML Test Framework

Committee Specification Version 1.0

OASIS ebXML Implementation, Interoperability and Conformance Technical Committee

07 March, 2003

Document identifier:

ebxml-iic-test-framework-10

Location:

http://www.oasis-open.org/committees/documents.php?wg_abbrev=ebxml-iic

Authors/Editors:

Steven Yung, Sun Microsystems <steven.yung@sun.com>
Prakash Sinha, IONA <prakash.sinha@iona.com>
Matthew MacKenzie, Individual <matt@mac-kenzie.net>
Hatem El Sebaaly, IPNet Solutions <hatem@ipnetsolutions.com>
Monica Martin, Sun Microsystems <monica.martin@sun.com>
Jacques Durand, Fujitsu <jdurand@fsw.fujitsu.com>
Michael Kass, NIST <michael.kass@nist.gov>
Eric VanLydegraf, Kinzan <ericv@kinzan.com>
Jeff Turpin, CycloneCommerce <jturpin@cyclonecommerce.com>
Serm Kulvatunyou, NIST <serm@nist.gov>

Contributors:

Christopher Frank < C.Frank@seeburger.de>

Abstract:

This document specifies ebXML interoperability testing specification for the eBusiness community.

Status:

This document has been approved as a committee specification, and is updated periodically on no particular schedule.

Committee members should send comments on this specification to the ebxml-iic@lists.oasis-open.org list. Others should subscribe to and send comments to the ebxml-iic-comment@lists.oasis-open.org list. To subscribe, send an email message to ebxml-iic-comment-request@lists.oasis-open.org with the word "subscribe" as the body of the message.

For more information about this work, including any errata and related efforts by this committee, please refer to our home page at http://www.oasis-open.org/committees/ebxml-iic.

Errata to this version:

None

Table of Contents

128

1 Introduction

1.1 Summary of Contents of this Document

This specification defines a test suite for ebXML Messaging basic interoperability. The testing procedure design and naming conventions follow the format specified in the Standard for Software Test Documentation IEEE Std 829-1998.

This specification is organized around the following topics:

- Interoperability testing architecture
- Test cases for basic interoperability
- Test data materials

1.2 Document Conventions

Terms in *Italics* are defined in the ebXML Glossary of Terms in the TestFramework specification [ebTestFramework]. Terms listed in ***Bold Italics*** represent the element and/or attribute content. Terms listed in `Courier` font relate to test data. Notes are listed in Times New Roman font and are informative (non-normative). Attribute names begin with lowercase. Element names begin with Uppercase.

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

- *MUST: This word, or the terms "REQUIRED" or "SHALL", means that the definition is an absolute requirement of the specification.*
- *MUST NOT: This phrase, or the phrase "SHALL NOT", means that the definition is an absolute prohibition of the specification.*
- *SHOULD: This word, or the adjective "RECOMMENDED", means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications MUST be understood and carefully weighed before choosing a different course.*
- *SHOULD NOT: This phrase, or the phrase "NOT RECOMMENDED", means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.*
- *MAY: This word, or the adjective "OPTIONAL", means that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation that does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation that does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides).*

1.3 Audience

The target audience for this specification is:

- The community of software developers who implement and/or deploy the ebXML Messaging Service (ebMS),

- The testing or verification authority, which will implement and deploy conformance or interoperability testing for ebXML Messaging implementations.

1.4 Caveats and Assumptions

It is assumed the reader has an understanding of communications protocols, MIME, XML, SOAP, SOAP Messages with Attachments and security technologies.

1.5 Related Documents

The following set of related specifications are developed independent of this specification as part of the ebXML initiative, they can be found on the OASIS web site (http://www.oasis-open.org).

- **ebXML Collaboration Protocol Profile and Agreement Specification [ebCPP]** – CPP defines one business partner's technical capabilities to engage in electronic business collaborations with other partners by exchanging electronic messages. A CPA documents the technical agreement between two (or more) partners to engage in electronic business collaboration. The MS Test Requirements and Test Cases will refer to CPA documents or data as part of their material, or context of verification.
- **ebXML Messaging Service Specification [ebMS]** – defines the messaging protocol and service for ebXML, which provide a secure and reliable method for exchanging electronic business transactions using the Internet.
- **ebXML Test Framework [ebTestFramework]**– describes the test architecture, procedures and material that are used to implement the MS Interoperability Test Suite, as well as the test harness for this suite.
- **ebXML MS Conformance Test Suite [ebMSConfTestSuite]**– describes the Conformance test suite and material for Messaging Services.
- **ebXML Registry Specification [ebRS]** – defines how one party can discover and/or agree upon the information the party needs to know about another party prior to sending them a message that complies with this specification. The Test Framework is also designed to support the testing of a registry implementation.
- **ebXML Business Process Specification Schema [BPSS]** – defines how two parties can cooperate through message-based collaborations, which follow particular message choreographies. The Test Framework is also designed to support the testing of a business process implementation.

1.6 Minimal Requirements for Conformance

An implementation of the Test Framework specified in this document MUST satisfy ALL of the following conditions to be considered a conforming implementation:

- It supports all the mandatory syntax, features and behavior (as identified by the [RFC2119] key words MUST, MUST NOT, REQUIRED, SHALL and SHALL NOT) defined in Part 1.1.1 – Document Conventions.
- It supports all the mandatory syntax, features and behavior defined for each of the components of the Test Framework.

It complies with the following interpretation of the keywords OPTIONAL and MAY: When these keywords apply to the behavior of the implementation, the implementation is free to support these behaviors or not, as meant in [RFC2119]. When these keywords apply to data and configuration material used by an

implementation of the Test Framework, a conforming implementation of the Test Framework MUST be capable of processing these optional materials according to the described semantics.

2 Principles and Methodology of Operations

2.1 General Objectives

The ebXML Test Framework is intended to support conformance and interoperability testing for ebXML specifications. It describes a testbed architecture and its software components, how these can be combined to create a test harness for each type of ebXML testing. It also describes the test material to be processed by this architecture, a mark-up language and format for representing test requirements, and test suites (set of Test Cases).

The Test Framework described here has been designed to achieve the following objectives:

- The Test Framework is a foundation for testing all ebXML architectural components such as Messaging, Registry, BPSS, CPA, and Core Components.

- Test Suites and Test Cases that are related to these standards, can be defined in a formal manner (including Test Steps and verification conditions). They can be automatically processed by the Test Framework, and their execution can easily be reproduced.

The harnessing of an ebXML implementation (or possibly several, e.g. in case of interoperability) with the Test Framework requires a moderate effort. It generally requires some interfacing work specific to an implementation, in the case no standard interface (API) has been specified. For example, the Test Service (a component of the Test Framework) defines Actions that will need to be called by a particular MSH implementation. Besides this kind of interfacing, no application code needs to be written.

- Several testbed configurations - or test harnesses - can be derived from the Test Framework, depending on the objectives of the testing. For example, MS conformance testing will include a particular combination (architecture) of some components of the Test Framework, while interoperability testing will require another set-up.

- Operating the Test Framework - or one of the test harnesses that can be derived from it – in order to execute a test suite, does not require advanced expertise in the framework internals, once the test suites have been designed. The tests should be easy to operate and to repeat with moderate effort or overhead, by users of the ebXML implementation(s) and IT staff responsible for maintaining the B2B infrastructure, without expertise in testing activity.

- Users can define new Test Suites and Test Cases to be run on the framework. For this, they will script their tests using the proposed test suite definition language or mark-up (XML-based) for Test Cases.

- A Test Suite (either for conformance or for interoperability) can be run entirely and validated from one component of the framework: the Test Driver. This means that all test outputs will be generated - and test conditions verified - by one component, even if the test harness involves several – possibly remote – components of the framework.

- The verification of each Test Case is done by the Test Driver at run-time, as soon as the Test Case execution is completed. The outcome of the verification can be obtained immediately as the Test Suite has completed, and a verification report be generated.

2.2 General Methodology

This specification only addresses the technical aspect of ebXML testing, and this section describes the portion of testing methodology that relates directly to the usage of the Test Framework. A more general test program for ebXML, describing a comprehensive methodology oriented toward certification, is promoted by the OASIS Conformance TC and is described in [ConfCertTestFrmk] (NIST). When conformance certification is the objective, the ebXML Test Framework should be used in a way that is compliant with a conformance certification model as described in [ConfCertModelNIST]. More general resources on Testing methodology and terminology can be found on the OASIS site (www.oasis-open.org), as well as at NIST (www.itl.nist.gov.)

This specification adopts the terminology and guidelines published by the OASIS Conformance Committee [ConfReqOASIS].

The Test Framework is intended for the following mode of operation, when testing for conformance or for interoperability. In order for a testing process (or validation process) to be conform to this specification, the following phases need to be implemented:

- Phase 1: **Test Plan** (RECOMMENDED). An overall test plan is defined, which includes a validation program and its objectives, the conditions of operations of the testing, levels or profiles of conformance or of interoperability, and the requirements for Candidate Implementations to be tested (context of deployment, configuration).

- Phase 2: **Test Requirements Design** (MANDATORY). A list of Test Requirements (also called Test Assertions) is established for the tested specification, and for the profile/level of conformance/interoperability that is targeted. These Test Requirements should refer to the specification document. Jointly to this list, it is RECOMMENDED that Specification Coverage be reported. This document shows, for each feature in the original specification, the Test Requirements items that address this feature. It also estimates to which degree the feature is validated by these Test Requirements items.

- Phase 3: **Test Harness Design** (MANDATORY). A Test Harness is defined for this particular test plan. It describes an architecture built from components of the Test Framework, along with operation instructions and conditions. In order to be conforming to this specification, a test harness MUST be described as a system that includes a Test Driver as specified in this document, and MUST be able to interpret conforming test suites.

- Phase 4: **Test Suite Design** (MANDATORY). Each Test Requirement from Phase 2 is translated into one or more Test Cases. A Test Case is defined as a sequence of operations (Test Steps) over the Test Harness. Each Test Case includes: configuration material (CPA data), message material associated with each Test Step, test verification condition that defines criteria for passing this test. All this material, along with particular operation directives, defines a Test Suite, as specified in Part II. In order to be conforming to this specification, a test suite needs to be described as a document (XML) conforming to part II of this specification.

- Phase 5: **Validation Conditions** (RECOMMENDED). Validation criteria are defined for the profile or level being tested, and expressed as a general condition over the set of results from the verification report of each Test Case of the suite. These validation criteria define the certification or “badging” for this profile/level.

- Phase 6: **Test Suite Execution** (MANDATORY). The Test Suite is interpreted and executed by the test Driver component of the Test Harness.

3 The Test Framework Components

The components of the framework are designed so that they can be combined in different configurations, or Test Harnesses.

We describe here two components that are central to the Test Framework:

- The Test Driver, which interprets Test Case data and drives Test Case execution.
- The Test Service, which implements some test operations (actions) that can be triggered by messages. These operations support and automate the execution of Test Cases.

These components interface with the ebXML Message Service Handler (MSH), but are not restricted to testing an MSH implementation.

3.1 The Test Driver

The Test Driver is the component that drives the execution of each step of a Test Case. Depending on the test harness, the Test Driver may drive the Test Case by interacting with other components in *connection mode* or in *service mode.*

- In connection mode, the Test Driver directly generates ebXML messages at transport protocol level – e.g. by using an appropriate transport adapter.
- In service mode, the Test Driver does not operate at transport level, but at application level, by invoking actions in the Test Service, which is another component of the framework. These actions will in turn send or receive messages to and from the MSH.

3.1.1 Functions

The primary function of the Test Driver is to parse and interpret the Test Case definitions that are part of a Test Suite, as described in the Test Framework mark-up language. Even when these Test Cases involve several components of the Test Framework, the interpretation of the Test Cases is under control of the Test Driver.

The Test Driver component of the ebXML Test Framework MUST have the following capabilities:

- **Self-Configuration -** Based upon supplied Test Case configuration parameters specified in the ebXML TestSuite.xsd schema (Appendix C), Test Driver configuration is done at startup, and MAY be modified at the TestCase and TestStep levels as well.
- **ebXML Message Construction –** Includes MIME, SOAP and ebXML portions of the message

- **Persistence of (Received) Messages –**received messages MUST persist for the life of a Test Case. Persistent messages MUST validate to the ebXMLMessageStore.xsd schema in Appendix D.
- **Parse and query persistent messages –** Test Driver MUST use XPath query syntax to query MIME, SOAP and ebXML persistent message content
- **Parse and query message payloads –** Test Driver MUST support XPath query syntax to query XML message payloads of persistent messages.
- **Controls the execution and workflow of the steps of a Test Case**. Some steps may be executed by other components, but their initiation is under control of the Test Driver.
- **Repeat previously executed Test Steps –** Test Driver MUST be capable of repeating previously executed Test Steps for the current Test Case.
- **Send messages** - Either directly at transport layer (e.g. by opening an HTTP connection), or by using Test Service actions.
- **Receive messages** - Either directly at transport layer, or by notification from Test Service actions.
- **Perform discreet message content validation –** Test Driver MUST be capable of performing discreet validation of Time, URI, Signature and the entire XML message
- **Perform discreet payload content validation –** Test Driver MUST be capable of performing discreet validation of Time, URI, Signature and an XML payload
- **Report Conformance Test Results –** Test Driver MUST generate an XML conformance report for all executed tests in the profile. Conformance reports MUST validate to the ebXMLTestReport.xsd schema in Appendix E.

A possible design that supports these functions is illustrated in Figure 1.

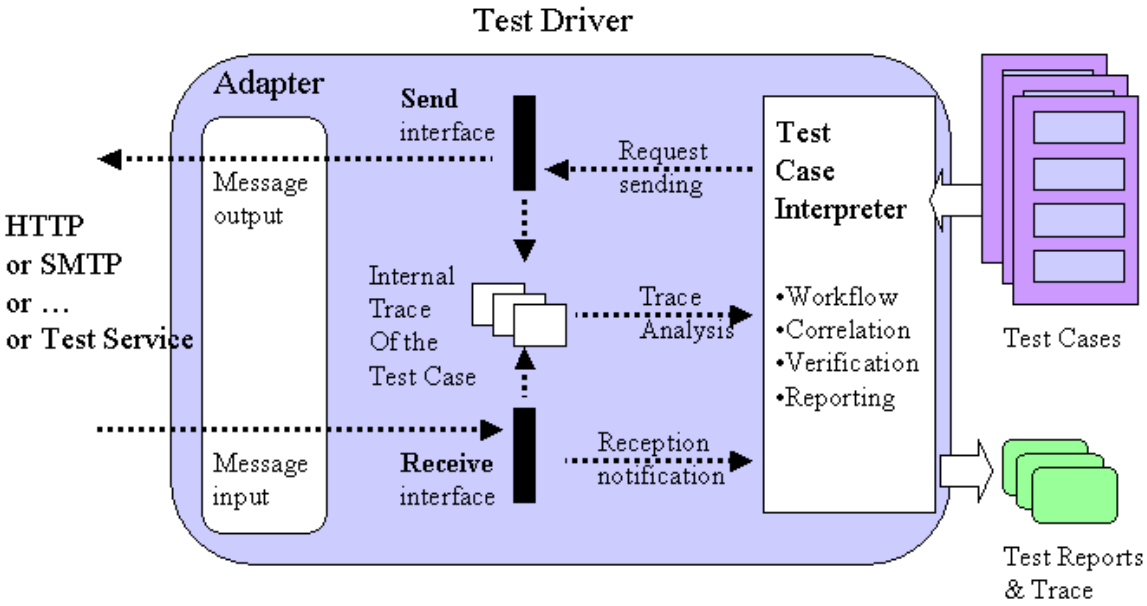


Fig 1. The Test Driver: Functions and Data Flows

1

3.1.2 Using the Test Driver in Connection Mode

The Test Driver MUST be able to control the inputs and outputs of an MSH at transport level. This can be achieved by using an embedded transport adapter. This adapter has transport knowledge, and can format message material into the right transport envelope. Independently from the way to achieve this, the Test Driver MUST be able to:

- Create a message envelope for the transports authorized by ebXML MS 2.0, and generate fully formed messages for this transport.
- Parse a message envelope for the transports authorized by ebXML MS 2.0, and extract header data from a message, as well as from the message payload in case it is an XML document.
- Open a message communication channel (connection) with a remote ebXML message handler. In that case the Test Driver is said to operate in connection mode.

When used in connection mode, the Test Driver is acting as a transport end-point that can receive or send messages with an envelope consistent with the transport protocol (e.g. HTTP or SMTP). The interaction between the MSH and the Test Service is of same nature as the interaction between the MSH and an application (as the Test Service simulates an application), i.e. it involves the MSH API, and/or a callback mechanism. Figure 2 illustrates how the Test Driver operates in connection mode.

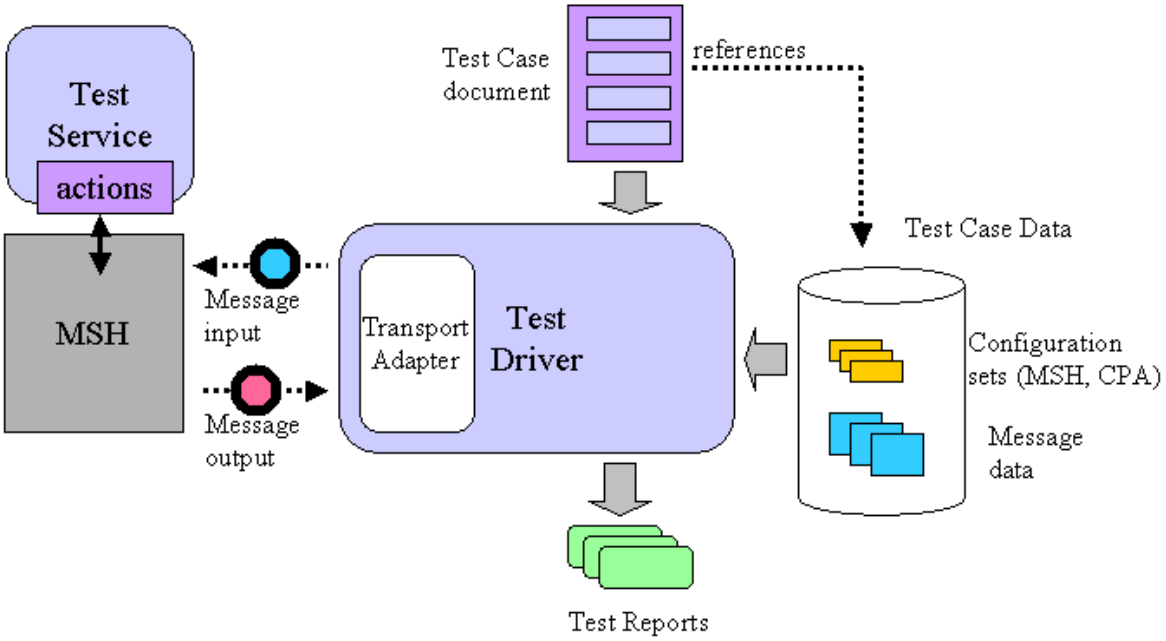


Fig 2. Test Driver used in Connection Mode

2

Figure 3 shows an example of conformance test harness with Test Driver used in connection mode.

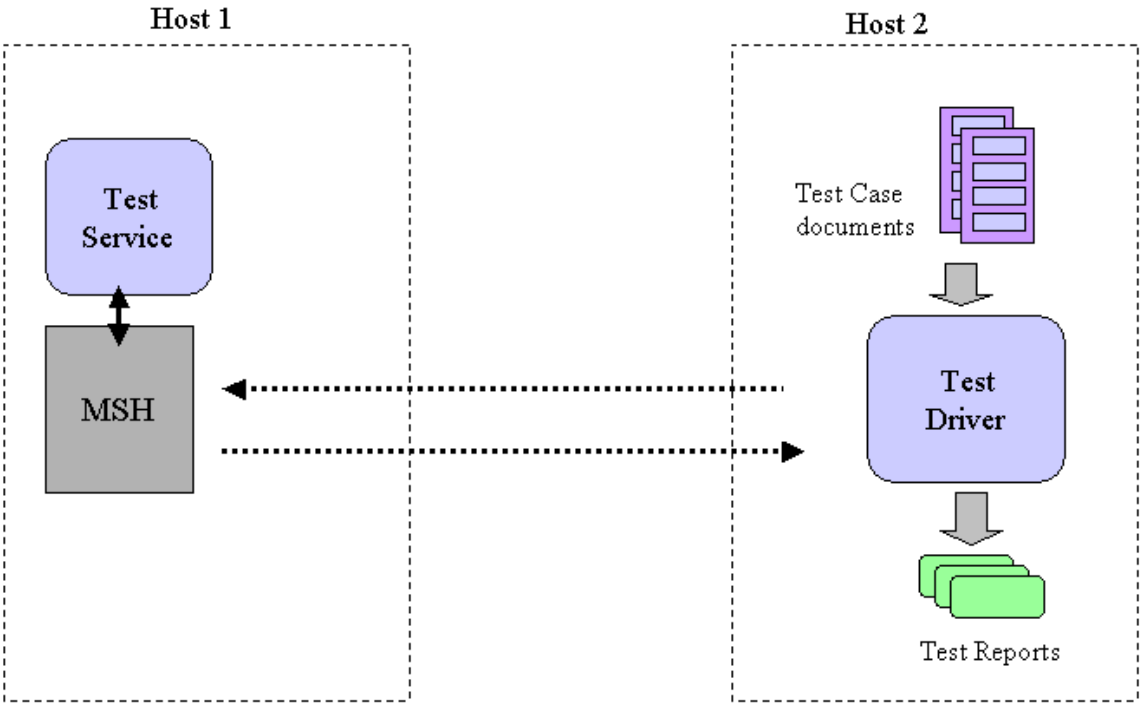


Fig 3. Example of Test Driver in Connection Mode: remote Conformance Testing of a Message Handler 3

3.1.3 Using the Test Driver in Service Mode

In this configuration, the Test Driver directly interacts with the Service/Actions of the Test Service component, without involving the transport layer, e.g. by invoking these actions via a software interface, in the same process space. This allows for controlling the Test Cases execution from the application layer (as opposed to the transport layer). Such a configuration is appropriate when doing interoperability testing - for example between two MSH implementations – and in particular, in situations where the transport layer should not be tampered with, or interfered with. The interactions with the Test Service will consist of:

- **Sending**: One action of the Test Service, the “Initiator”, serves as a channel to send requests to the MSH it has been interfaced with. This action – normally triggered by received messages – also MUST provide an interface at application level. When invoked by a call that contains message data, the action generates a sending request to the MSH API for this message.

- **Receiving**: As all actions of the Test Service can participate in the execution of a Test Case (i.e. of its Test Steps), the Test Driver needs to be aware of their invocation by incoming messages. Each of these actions will notify the Test Driver through its “Receive” interface, passing received message data, as well as response data. This way, the Test Driver will build an internal trace (or state) for the Test Case execution, and will be able to verify the test based on this data.

The Test Driver MUST support the above communication operations with the Test Service. This may be achieved by using an embedded Service Adapter to bridge the sending and receiving functions of the Test Driver, with the Service/Action calls of the Test Service. Figure 4 illustrates how the Test Driver operates with a Service Adapter.

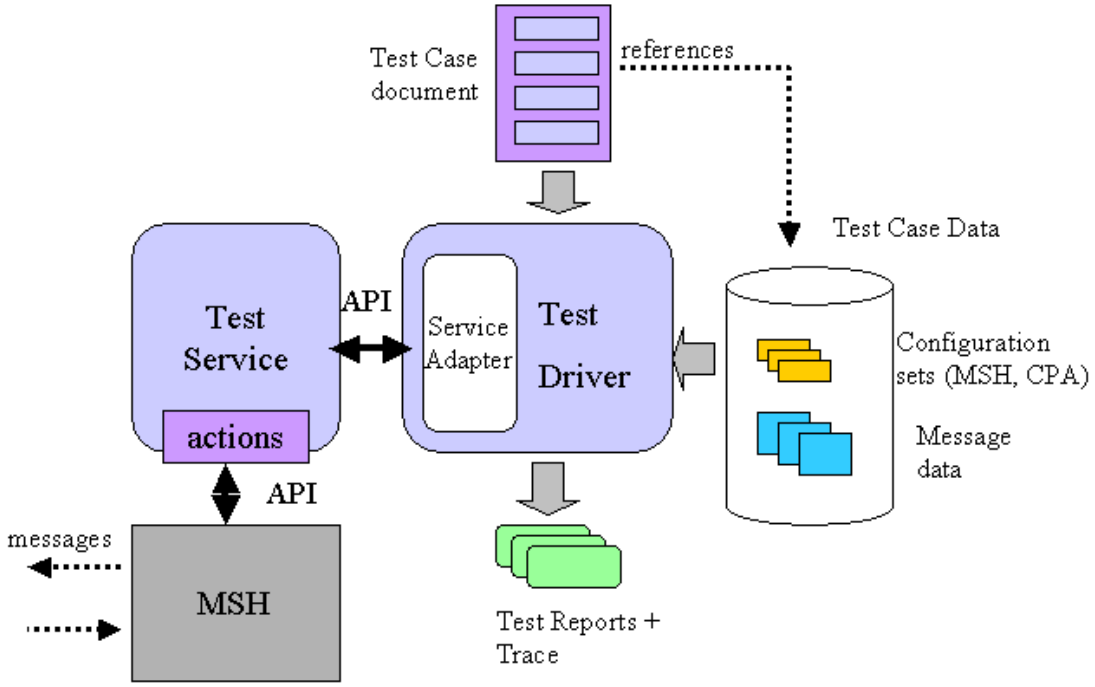


Fig 4. Test Driver used in Service Mode

4

This design allows for a minimal exposure of the MSH-specific API, to the components of the Test Framework. The integration code that needs to be written for connecting the MSH implementation is then restricted to an interface with the Service/Actions defined by the framework. Neither the Test Driver, nor the Service Adapter, need to be aware of the MSH-specific interface. An example of test harness using the Test Driver in Service Mode is shown in Figure 5.

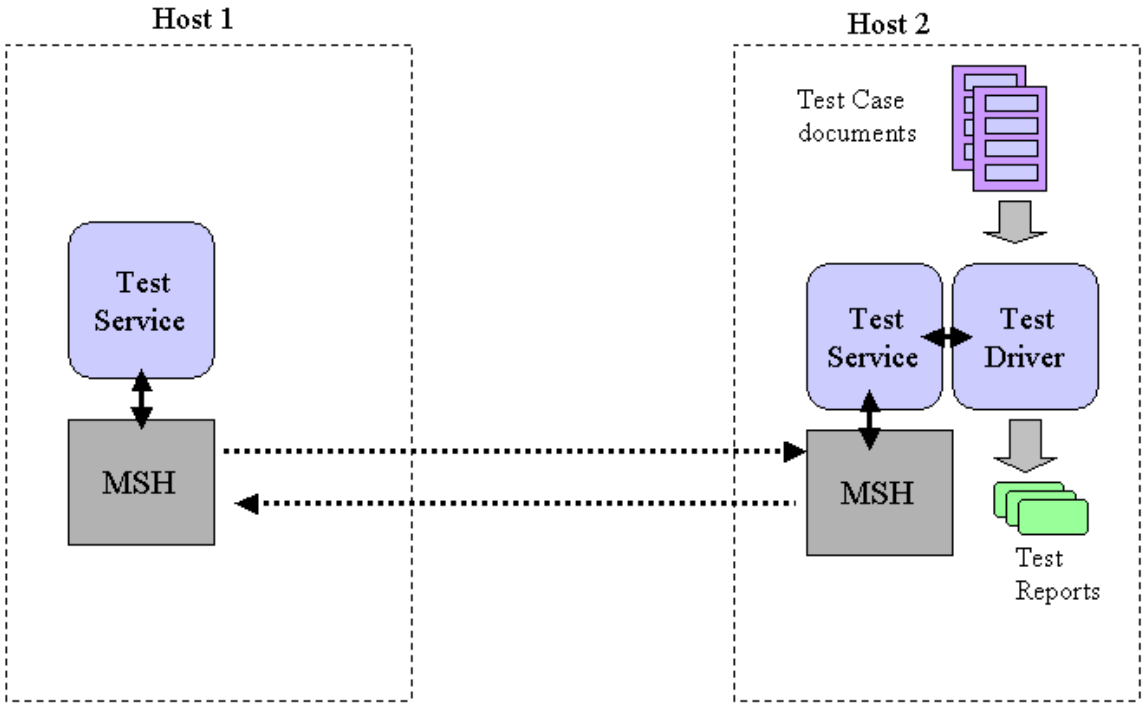


Fig 5. Example of Test Driver in Service Mode: Point-to-Point Interoperability Testing of Message Handlers 5

3.2 The Test Service

3.2.1 Functions and Interactions

The Test Service defines a set of Actions that are useful for executing Test Cases. The Test Service represents the application layer for a message handler. It receives message content and error notifications from the MSH, and also generates requests to the MSH, which normally are translated into messages being sent out. The Test Actions are predefined, and are part of the Test Framework (i.e. not user-written). Test Service and Actions will map to the Service and Action header attributes of ebXML messages generated during the testing.

The Test Service name is: `urn:ebXML:iic:test`.

Figure 6 shows the details of the Test Service and its interfaces.

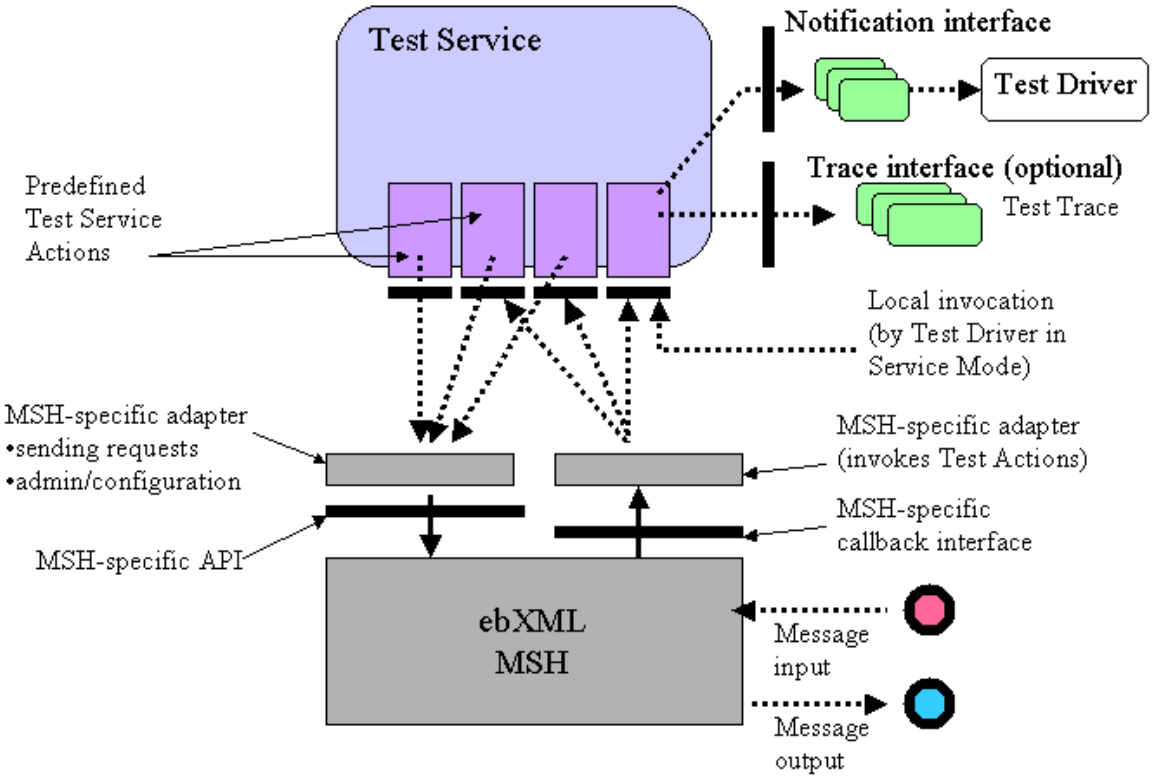


Fig 6. The Test Service and its Interfaces 6

The functions of the Test Service are:

- To implement the actions which map to Service / Action fields in a message header. The set of test actions which are pre-defined in the Test Service will perform diverse functions, which are enumerated below:
- To notify the Test Driver of incoming messages. This only occurs when the Test Service is deployed in *reporting mode*, which assumes it is coupled with a Test Driver.
- To perform some message processing, e.g. compare a received message payload with a reference payload (or their digests).
- To send back a response to the MSH. Depending on the action invoked, the response may range from a pre-defined acknowledgment to a specific message previously specified.
- Optionally, to generate a trace of its operations, in order to help trouble-shooting, or for reporting purpose.

Although the Test Service simulates an application, it is part of the Test Framework, and does not vary from one test harness to the other. However, in order to connect to the Test Service, a developer will have to write wrapper code to the Test Service/Actions that is specific to the MSH implementation that needs to be integrated. This proprietary code is expected to require a minor effort, but is necessary as the

API and callback interfaces of each MSH are not specified in the [ebMS] standard and is implementation-dependent.

3.2.2 Modes of Operation of the Test Service

The Test Service can operate in two modes:

- **Reporting mode**: in that mode, the actions of the Test Service instance, when invoked, will send a notification to the Test Driver. The Test Driver can then be aware of the workflow of the test case. There are actually two cases of **reporting** mode:
  - **Local Reporting Mode:** The Test Driver is installed on the same host as the Test Service, and executes in the same process space. The notification uses the *Receive* interface of the Test Driver, which is operating in service mode.
  - **Remote Reporting Mode:** The Test Driver is installed on a different host than the Test Service. The notification is done via messages to the Test Driver, which is generally operating in connection mode. Remote message notifications are identified by the Test Driver through the "Notify" Action name and "`urn:ebXML:iic:test`" Service name, in the header of a received ebXML message.

- **Loop mode**: in that mode, the actions of the Test Service instance, when invoked, will NOT send a notification to the Test Driver. The only interaction of the Test Service with external parties, is by sending back messages via the message handler

Except for the notification, which may or may not take place, the actions operate similarly in both reporting and loop modes, unless specified otherwise. In other words, the mode of operation does not normally affect the logic of the action. The action may send a response message, to the requesting party via the "response URL". In general, the response URL is the same as the requestor URL.

Figure 7 shows a test harness with a Test Driver in connection mode, controlling a Test Service (Host 1) in remote reporting mode. The other Test Service (Host 3) is operating in loop mode. This configuration is used when the test cases are controlled from a third party test center, when doing interoperability testing. The test center may also act as a Hub, and be involved in monitoring the traffic between the interoperating parties.

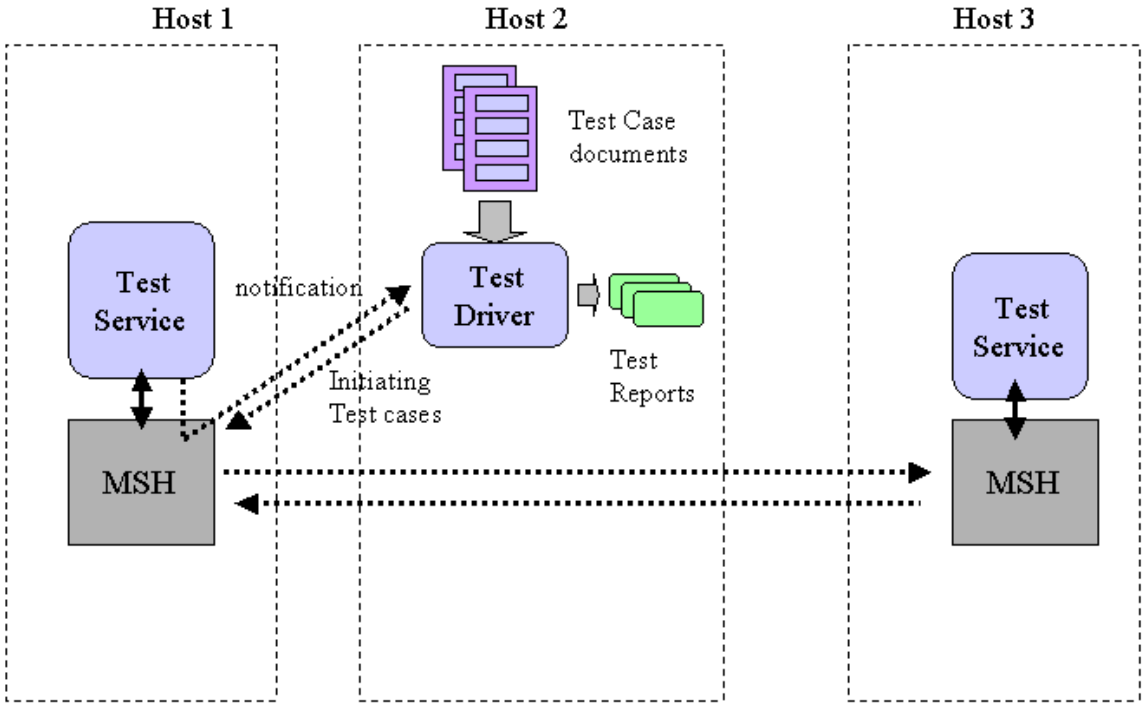


Fig 7. Example of Test Service in Remote Reporting Mode: The InteroperabilityTest Center model 7

3.2.3 Parameters of the Test Service

The Test Service has only two parameters that can be modified by action invocation (and therefore represent its state):

- Operation mode (either reporting or loop)
- Response URL (destination for response messages)
- Notification URL (destination for notification messages, if applicable)

In addition, a Test Service instance is identified by an ID that will be reported in some response messages. The three parameters above can be set by invoking the Configurator action described below. In a test harness where an interoperability test suite involves two parties, the test suite will need to be executed twice - alternatively driven from each party. In that case, each Test Service instance will alternatively be set to a reporting mode, while the other will be set to loop mode. These settings can be done remotely by sending messages to the Configurator action.

Except for these parameters, the Test Service is stateless.

3.2.4 The Actions of the Test Service

The actions described here are standard to the Test Service, and should suffice in supporting most Test Cases. These actions map to the Service/Action field of an ebXML message, and will be triggered on reception of such messages.

3.2.4.1 Common Functions

Some functions are common to several actions, in addition to the specific functions they fulfill. These common functions are:

- **Generate a response message**. Response messages are destined to the Response URL (see 3.2.3). They also specify a Service/Action, as they are usually intended for another Test Service although in case the Response URL directly points to the Test Driver in connection mode, Service/Action will not have the regular MSH semantics.
- **Notify the Test Driver**. This assumes the Test Service is coupled with a Test Driver. In that configuration, the Test Service is in reporting mode. The reporting is done by a message (sent to the Notification URL) when in remote reporting mode, or by a call to the Receive interface when in local reporting mode.

3.2.4.2 Test Service Actions

The standard test actions are:

3.2.4.2.1 **Mute** action

Reporting/Loop Mode Action Description: This is a “dummy” action, which does not generate any response message back. Such an action is used for messages that do not require any effect, except possibly to cause some side-effect in the MSH, like generating an error.

Response Destination: None

In Reporting Mode: The action will notify the associated Test Driver. The notification containing the received header and payload(s) material, will be done via the Receive interface, if in local reporting mode, or with a message with Service / Action fields set to “`urn:ebXML:iic:test`”**/ “Notify”,** if in remote reporting mode. The notification will report the action name (“mute”) and the instance ID of the Test Service.

3.2.4.2.2 **Dummy** action

Reporting/Loop Mode Action Description: This is a “dummy” action, used by messages that do not need a specific response. On invocation, this action will however generate a pre-canned response message back

(no payload, simplest header with no extra-features), with no dependency on the received message, except for the previous MessageID (for correlation) in the RefToMessageId header attribute.

Response Destination: a message with a **Mute** action element is sent to the Test Component (Test Driver or Service) associated with the Response URL. This notice serves as proof that the message has been received, although no assumption can be made on the integrity of its content.

In Reporting Mode: The action will also notify the associated Test Driver. The notification containing the received header and payload(s) material, will be done via the Receive interface, if in local reporting mode, or with a message with Service / Action fields set to "`urn:ebXML:iic:test`"**/ "Notify",** if in remote reporting mode. The notification will report the action name ("Dummy") and the instance ID of the Test Service.

3.2.4.2.3 Reflector

Reporting/Loop Mode Action Description: On invocation, this action generates a response to a received message, by using the same message material, with minimal changes in the header:

- Swapping of the to/from parties so that the "to" is now the initial sender.
- Setting RefToMessageId to the ID of the received message.
- Removing AckRequested or syncReply elements if any.
- All other header elements (except for time stamps) are unchanged. The conversation ID remains unchanged, as well as the CPAId. The payload is the same as in the received message, i.e. same attachment(s).

Response Destination: a message with a **Mute** action element is sent to the Test Component (Test Driver or Service) associated with the Response URL. This action acts as a *reflector* for the initial sending party

In Reporting Mode: The action also notifies the associated Test Driver. The notification containing the received header and payload(s) material, will be done via the Receive interface, if in local reporting mode, or with a message with Service / Action fields set to "`urn:ebXML:iic:test`"**/ "Notify",** if in remote reporting mode. The notification will report the action name ("reflector") and the instance ID of the Test Service.

3.2.4.2.4 **Initiator** action

Reporting/Loop Mode Action Description: On invocation, this action generates a new message, totally unrelated to the header data of the enclosing ebXML message. The new message material (payload and header) is provided in a pre-convened way in the payload of the received message. The header of the new message can be anything that is specified. For example, this action would be used to generate a "first" message of a new conversation, different from the conversation ID specified in the invoking message. Note: unlike in the Reflector action, MSH-controlled header attributes will not be determined by the invoking message header (messageID, RefToMessageId, timestamps...). So if the response needs to refer to the previous MessageID (for correlation), the RefToMessageId must be explicitly pre-set in the message material.

Response Destination: any service/action of the sender, specified with message material (by default: a message with a **Mute** action element is sent to the Test Component (Test Driver or Service) associated with the Response URL.)

In Reporting mode: In addition to generating the message, the action also notifies the associated Test Driver. The notification containing the received header and payload(s) material, will be done via the

Receive interface, if in local reporting mode, or with a message with Service / Action fields set to "`urn:ebXML:iic:test`" **/ "Notify",** if in remote reporting mode. The notification will report the action name ("initiator") and the instance ID of the Test Service.

3.2.4.2.5 **PayloadVerify** action

Reporting/Loop Mode Action Description: On invocation, this action will compare the payload(s) of the received message, with the expected payload. Instead of using real payloads, to be pre-installed on the site of the Test Service, it is RECOMMENDED that a digest (or signature) of the reference payloads (files) be pre-installed on the Test Service host. The PayloadVerify action will then calculate the digest of each received payload and compare with the reference digests. This action will test the service contract between application and MSH, as errors may originate either on the wire, or at every level of message processing in the MSH until message data is passed to the application. The action responds with a response message to the party associated with the Response URL, reporting the outcome of the comparison. The previous MessageID is reported (for correlation) in the RefToMessageId header attribute of the response. The previous ConversationId is also reported. The payload message will contain a verification status notification for each verified payload

The XML format used by the response message is described in the section 7.1.12 ("Service Messages").

Response Destination: a message is sent with a **Mute** action element to the Test Component (Test Driver or Service) associated with the Response URL.

In Reporting mode: Action will also notify the associated Test Driver. The notification containing the received header and payload(s) material, will be done via the Receive interface, if in local reporting mode, or with a message with Service / Action fields set to "`urn:ebXML:iic:test`" **/ "Notify",** if in remote reporting mode.

3.2.4.2.6 ErrorAppNotify action

Reporting/Loop Mode Action Description: This action will capture specific error notifications from the MSH to its using application. It is not triggered by reception of an error message, but it is directly triggered by the internal error module of the MSH local to this Test Service. If the MSH implementation does not support such direct notification of the application (e.g. instead, it writes such notifications to a log), then an adapter needs to be written to read this log and invoke this action whenever such an error is notified.

Such errors fall into two categories:

- MSH errors that need to be directly communicated to its application – and not to any remote party, e.g. failure to send a message (no Acks received after maximum retries).
- In case an MSH generates regular errors with a severity level set to "Error" – as opposed to "Warning" – the MSH is supposed to (SHOULD) also notify its application. The ErrorAppNotify action is intended to support both types of notifications.

Response Destination: A response message containing the verification status notification (see "Service Messages" in 7.1.12) is sent only when in loop mode, with a **Mute** action element to the Test Component (Test Driver or Service) associated with the Response URL.

In Reporting mode: Action will notify an error to the associated Test Driver The notification containing the received ErrorList document only (not the received message content), will be done via the Receive interface, if in local reporting mode, or with a message with Service / Action fields set to "`urn:ebXML:iic:test`" **/ "Notify",** if in remote reporting mode.

3.2.4.2.7 **ErrorURLNotify** action

Reporting/Loop Mode Action Description: This action will capture error messages, assuming that an adapter has been written for invoking this action. The adapter must have same URI as the ErrorURI specified in the CPA. The adapter will pass the entire message as is (in its ebXML envelope) to the action. The action extracts the ErrorCode and Severity elements, and sends then an error notification message back to the originator, when operating in loop mode only. The action will make such error notifications visible to the other party (generally the driver party), by generating a "report" message to the Response URL.

The XML format used to both the received and response message payload for this action is described in section 8.1.2, the Test Message Schema.

Response Destination: When in loop mode, generates a verification status notification (see "Service Messages" in 7.1.12) with a **Mute** action element to the Test Component (Test Driver or Service) associated with the Response URL.

In Reporting mode: Action only notifies the associated Test Driver. The notification containing the received header and payload(s) material, will be done via the Receive interface, if in local reporting mode, or with a message with Service / Action fields set to "`urn:ebXML:iic:test`" **/ "Notify",** if in remote reporting mode. The notification will report the action name ("ErrorURLNotify") and the instance ID of the Test Service.

3.2.4.2.8 **Configurator** action

Reporting/Loop Mode Action Description: This action is called to either dynamically (re)configure the receiver party, or to verify that the receiver party has the right configuration set-up. Configuration may concern:

- MSH internals assumed by a Test Case (if applicable),
- CPA set-up assumed by a Test Case,
- Test Service parameters (e.g. ID, response-URL, mode of operation). In the case of CPA, the action can verify that the collaboration agreement for a conversation related to a Test Case or a set of Test Cases is available. If the payload contains a CPAId, this action will verify that the corresponding CPA is accessible. The previous MessageID is reported (for correlation) in the RefToMessageId header attribute of the response.
- Predefined digests of payloads to be used in Test Cases. These digests or signatures will be then used as references for comparing digests from received payloads. Such comparison will be done by the PayloadVerify action.

The XML format used by the configuration message and its response message is described in the section 7.1.12 ("Service Messages").

Response Destination of response: a message with a **Mute** action element is sent to the Test Component (Test Driver or Service) associated with the ResponseURL.

In Reporting mode: Action notifies the associated Test Driver. The notification containing the received header and payload(s) material, will be done via the Receive interface, if in local reporting mode, or with a

message with Service / Action fields set to "`urn:ebXML:iic:test`" **/ "Notify",** if in remote reporting mode. The notification will report the action name ("configurator") and the instance ID of the Test Service.

Note: The above actions are specific to the Test Service, and are not supported by the Test Driver.

3.2.4.3 Integration with an Implementation

As mentioned before, the actions above are predefined and part of the Test Framework, and will require some integration code with the MSH implementation, in form of three adapters, to be provided by the MSH development (or user) team. These adapters are:

(1) **Reception adapter**, which is specific to the MSH callback interface. This code allows for invocation of the actions of the Test Service, on reception of a message.

(2) **MSH control adapter**, which will be invoked by some Test Service actions, and will invoke in turn the MSH-specific Message Service Interface (or API). Examples of such invocations are for sending messages (e.g. by actions which send response messages), and MSH configuration changes (done by the Configurator action).

(3) **Error URL adapter**, which is actually independent from the candidate MSH. This adapter will catch error messages, and invoke the **ErrorURLNotify** action of the Test Service. If the Test Service is in reporting mode, the Test Driver is notified of this error message.

3.3 Executing Test Cases

A Test Suite contains a sequence of Test Cases. Each Test Case is intended to verify that an implementation fulfills a requirement item (or a set of items) of the specification.

3.3.1 Test Case as a Sequence of Test Steps

:

A Test Case is a sequence of Test Steps. A Test Step is an aggregate of one or more operations performed by a single component of the test harness. A Test Step usually involves a single message sending or receiving operation, plus some message data processing operations, like checking a condition

on message header. A Test Step may also include conditional actions that are a basis for the execution of the assertion within the Test Step itself.

A Test Case instance is an execution of a particular Test Case, identified by some specific message attribute values. For example, two instances of the same Test Case will be distinguished by distinct MessageID values in the generated messages. An example of a sequence of Test Steps associated with an MS Conformance Test Case is:

Step 1: Test driver sends a sample message to the Reflector action of the Test Service. Message header data is obtained from message header declaration, and message payload from file ABC.

Step 2: Test driver receives the response message and adds it to the stored sequence for this Test Case instance (correlation with Step 3 is done based on the RefToMessageID attribute, which should be identical to the MessageId of Step 3.)

Step 3: Test driver verifies the test condition on response message, for example that the SOAP envelope and extensions are well-formed.

3.3.2 Related Message Data and Message Declarations

Some Test Steps will require message data. This message data MUST be specified using a Message Declaration (Section 7), which is an XML-based script. Header content is scripted in the message declaration by using XML and XPath expressions. The message envelope is also described in the same way. Payload material is not included in the messages declaration, but referenced by it. The script that describes a test step may also include operations that allow for extracting a payload from or for adding a payload to a message. The Test Driver MUST be capable of interpreting these scripts in order to:

- Assemble a message from script material and referenced payloads.
- Analyze and select a received message based on header and envelope content (as well as based on payload content if the payload is in XML).

3.3.3 Related Configuration Data

Test Cases will be executed under a pre-defined agreement, as defined in CPA [ebXML CPA]. This agreement will configure the ebXML Candidate Implementations involved in the testing, or the collaborations that execute on these implementations. Each Test Case will therefore reference a Test Configuration document.

- **Test Configuration document**: it contains (1) a CPA (or CPA-like) document, (2) configuration data for the ebXML implementation(s) involved, expressed at an abstract level and expected to be general enough to most implementations, even if not specified.

Figure 8 illustrates how a Test Case references message data.

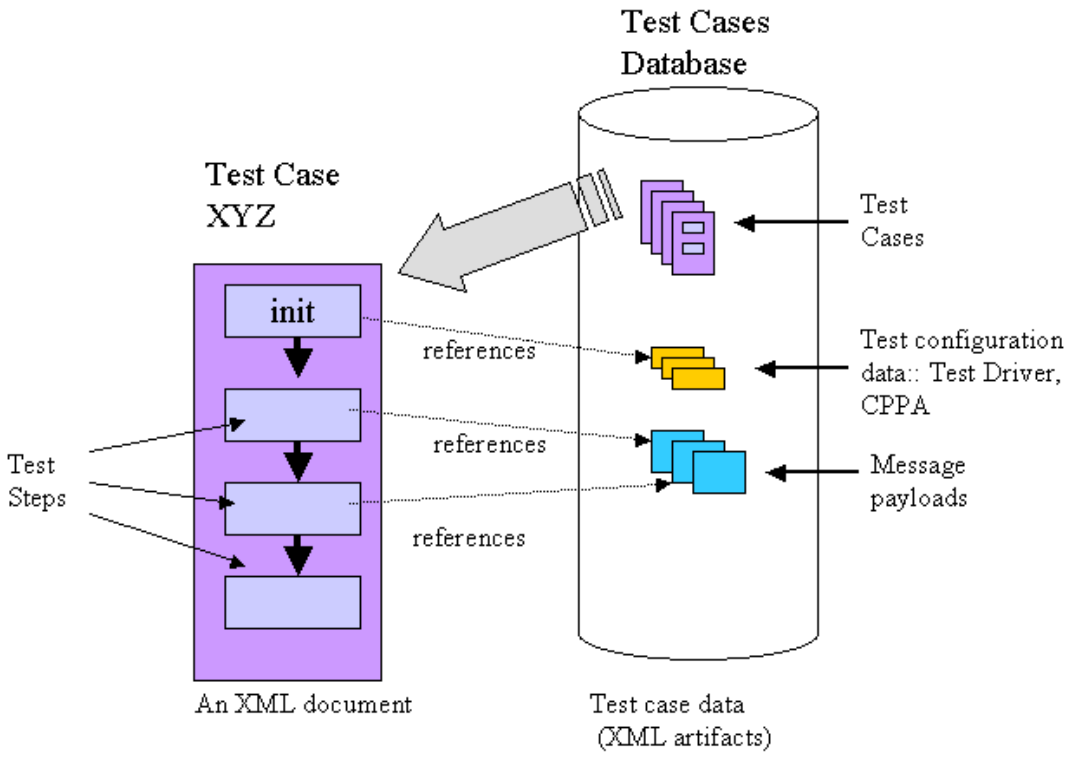


Fig 8. Test Case Document and Database

9

Part II: Test Suite Representation

4 Test Suite

4.1 Conformance vs. Interoperability Test Suite

We distinguish two types of test suites, which share similar document schemas and architecture components, but serve different purposes:

- **Conformance Test Suite**. The objective is to verify the adherence or non-adherence of a Candidate Implementation to the target specification. The test harness and Test Cases will be designed around a single (candidate) implementation. The suite material emphasizes the target specification, by including a comprehensive set of Test Requirements, as well as a clear mapping of these to the original specification (e.g. in form of an annotated version of this specification).

- **Interoperability Test Suite**. The objective is to verify that two implementations (or more) of the same specification, or that an implementation and its operational environment, can interoperate according to an agreement or contract (which is compliant with the specification, but usually restricts further the requirements). These implementations are assumed to be conforming (i.e. have passed conformance tests or have achieved the level of function of such tests), so the reference to the specification is not as important as in conformance. Such a test suite involves two or more Candidate Implementations of the target specification. The test harness and Test Cases will be designed in order to drive and monitor these implementations.

A conformance test suite is composed of:

- One or more **Test Profile** documents (XML). Such documents represent the level or profile of conformance to the specification, as verified by this Test Suite.
- Design of a **Test Harness** for the Candidate Implementation that is based on components of the ebXML IIC Test Framework.
- A **Test Requirements** document. This document contains a list of conformance test assertions that are associated with the test profile to be tested.
- An **annotation** of the target specification, that indicates the degree of Specification Coverage for each specification feature or section, that this set of Test Requirements provides.
- A **Test Suite** document. This document implements the Test Requirements, described using the Test Framework material (XML mark-up, etc.)

An Interoperability Test Suite is composed of:

- One or more **Test Profile** documents (XML). Such documents represent a set of features specific to a particular functionality, represented in a Test Suite through Test Cases that only test those particular features, and hence, that profile.
- Design of a **Test Harness** for two or more interoperating implementations of the specification that is based on components of the ebXML Test Framework.
- A **Test Requirements** document. This document contains a list of test assertions associated with this profile (or level) of interoperability.
- A **Test Suite** document. This document implements the Test Requirements, described using the Test Framework material (XML mark-up, etc.)

4.2 The Test Suite Document

The Test Suite XML document is a collection of Test Driver configuration data, documentation and executable Test Cases.

- **Metadata** provides documentation used by the Test Driver to generate a Test Report for all executed Test Cases.
- **Configuration data** provide basic Test Driver parameters used to modify the configuration of the Test Driver to accurately perform and evaluate test results. It also contains configuration data for the candidate ebXML implementation(s).
- **Message data** is a collection of pre-defined XML payload messages that can be referenced for inclusion in an ebXML test message.
- **Test Cases** are a collection of discrete Test Steps. Each Test Step can execute any number of test Operations (including sending, receiving, and examining returned messages). An ebXML Test Suite document MUST validate against the ebXMLTestSuite.xsd file in Appendix C.
- **Payloads** provide XML and non-XML content for use as material for test messages, as well as message data for Test Services linked to the Test Driver.

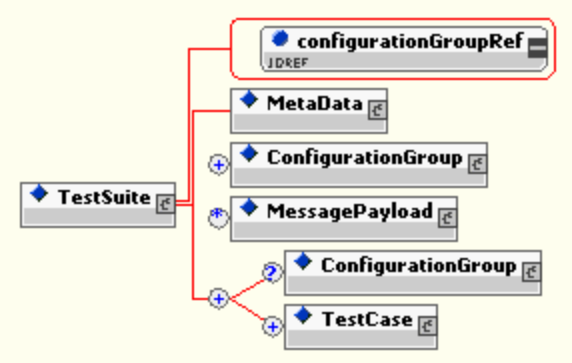


Figure 9 – Graphic representation of basic view of ebXMLTestSuite.xsd schema

Definition of Content

Name	Description	Default Value From Test Driver	Required/Optional
TestSuite	Container for all configuration, documentation and tests		Required
Metadata	Container for general documentation of the entire Test Suite		Required
ConfigurationGroup	Container for configuration of the Test		Required

	Driver and /or MSH		
MessagePayload	XML Payload message for inclusion in a Test Case		Optional
ConfigurationGroup	Container for modifications to base ConfigurationGroup		Optional
TestCase	Container for an individual Test Case		Required

Test Suite Metadata

Documentation for the ebXML MS Test Suite is done through the Metadata element. It is a container element for general documentation.

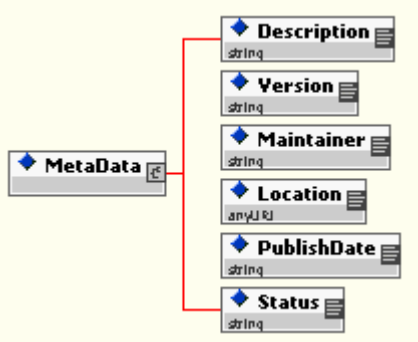


Figure 10 – Graphic representation of expanded view of the Metadata element

Definition of Content

Name	Description	Default Value From Test Driver	Required/Optional
Description	General description of the Test Suite		Required
Version	Version identifier for Test Suite		Required
Maintainer	Name of person(s) maintaining the Test Suite		Required
Location	URL or filename of this test suite		Required
PublishDate	Date of publication		Required
Status	Status of this test suite		Required

4.2.1 The ConfigurationGroup

The ConfigurationGroup contains configuration data for modifying the content of test messages sent by the Test Driver (when in Connection Mode) or the MSH (when the Test Driver is in Service Mode) In addition, three of the parameters have functions beyond defining message content. The “Mode” parameter toggles the mode of the Test Driver between Connection Mode and Service Mode. CPAId is used by the Test Driver (in Service Mode) to configure its interfaced MSH through its CPAId reference. PayloadDigests provides a list of payload identifiers and their MDA-5 digest values for payload content verification by the Test Driver or Test Service.

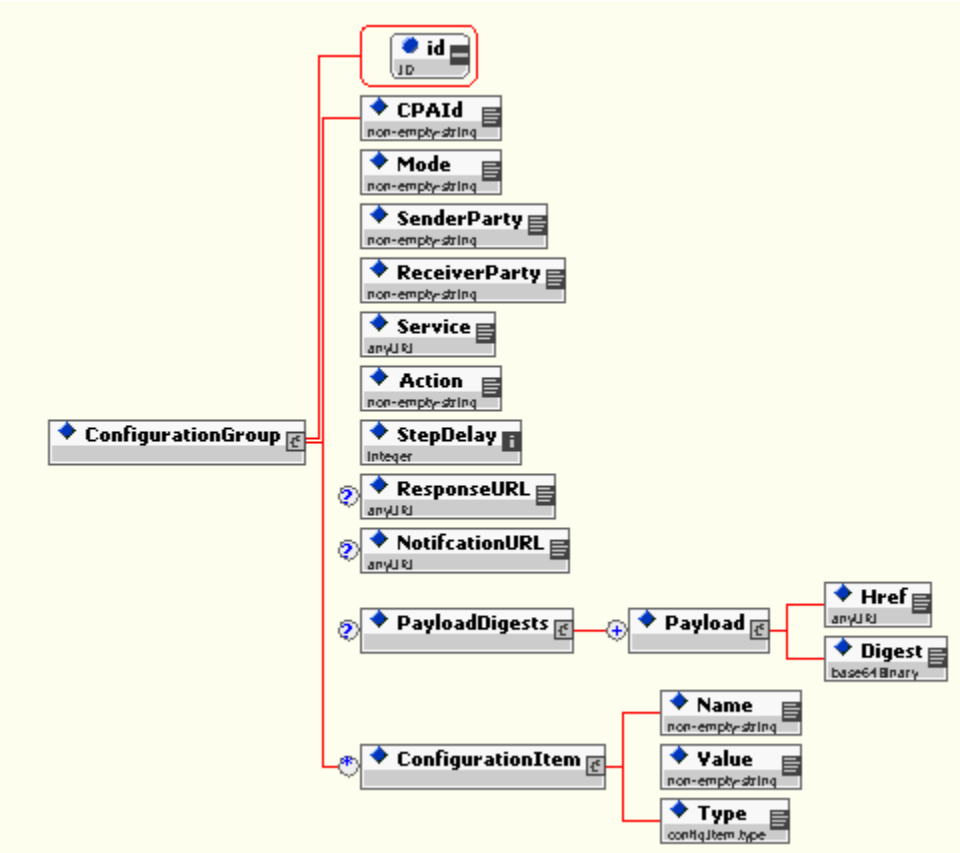


Figure 11 – Graphic representation of expanded view of the BaseConfigurationGroup element

Definition of Content

Name	Description	Default Value From Test Driver	Required/Optional
ConfigurationGroup	Container Test Driver/MSH configuration data		Required

CPAId	Unique identifier matching one of the testing CPA's in the Conformance or Interoperability Test Suite. Inserted inside outgoing messages, as content for the CPAId element in an ebXML MessageHeader. Value is also used to configure MSH when in "service" mode.		Optional
Mode	One of two types, "driver" (interfaced to MSH) or "non-driver" (standalone)	Non-driver	Required
SenderParty	Default identifier used in message header From/PartyId		Required
ReceiverParty	Default identifier used in message header To/PartyId		Required
Service	Default Service to be inserted into outgoing message Service element content		Required
Action	Default Service Action to be inserted into outgoing messages Action element content		Required
StepDelay	Milliseconds of delay between execution of the last Test Step and the current Test Step		Required
ResponseURL	Parameter defining the URL for the Test Service to send response messages to		Optional
NotificationURL	Parameter defining the location for the Test Service to send notification messages to		Optional
PayloadDigests	Container for one or more message payload identifiers with corresponding computed digest values, used by Test Driver to verify received message payload content		Optional
Payload	Container for id and digest value pair		Required
href	Identifier (CID) for message payload to be verified against Digest value		Required
Digest	Pre-computed MDA-5 digest value to be used by Test Driver to verify integrity of received message payload		Required
ConfigurationItem	Container for individual name/value pair used by the Test Driver for configuration or possibly for message payload content construction		Optional
Name	Name for the ConfigurationItem		Required
Value	Value of the ConfigurationItem		Required
Type	Type of ConfigurationItem (namespace or parameter)		Required

4.2.1.1 Multiple Configuration Declarations within the Test Suite

It is possible to dynamically modify the configuration of the Test Driver or MSH through the course of execution of the Test Suite. After initial configuration is set through the required configurationGroupRef attribute of the TestSuite element, configuration can be modified at the Test Case and Test Step levels through the introduction of a new configurationGroupRef attribute value.

The scope of the configuration change is hierarchical, and exists only for the definition of the current Test Suite, Case or Step. Upon completion, of that test component, configuration reverts to that previously defined at the higher level. This hierarchy is illustrated in the figure below.

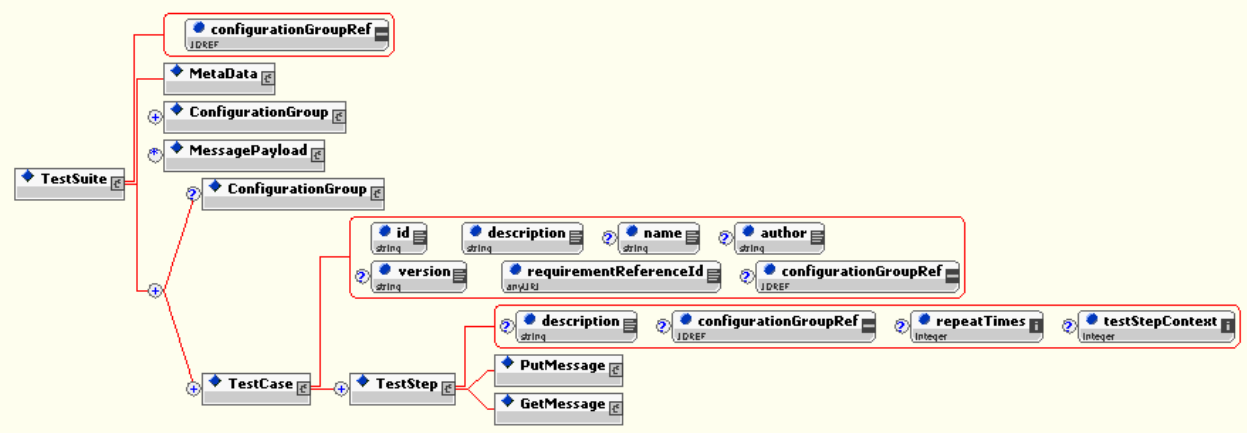


Figure 12 – Graphic representation of hierarchical use of the ConfigurationGroup via reference

4.2.1.2 Precedence Rules for Test Driver/MSH configuration

In order to generate messages correctly, the Test Driver MUST follow the precedence rules for interpreting a Configuration Group declaration. The precedence rules are:

Certain portions of an ebXML message are auto-generated by the Test Driver at run-time, unless they are overridden explicitly in the Message Declaration. These include:

- ConversationId
- MessageId
- Timestamp

In addition, other message content, such as the “soap:mustUnderstand” attribute and value, the ebXML “version” attribute of message header extension elements and other content is also auto-generated by the Test Driver (in Connection Mode) or interfaced MSH (in Service Mode) for every Message Declaration. These values can also be overridden through explicit declaration.

If there is no explicit declaration of an element or attribute value in the Message Declaration content, and it is not auto-generated by the Test Driver, and it is a required element/attribute value, then the Configuration Group value MUST be provided in the Message Declaration.

5 Test Requirements

5.1 Purpose and Structure

The next step in designing a test suite is to define Test Requirements. This material, when used in a conformance testing context, is also called Test Assertions in NIST and OASIS terminology (see definition in glossary in Appendix).

When used for conformance testing, each Test Requirement defines a test item to be performed, that covers a particular requirement of the target specification. It rewords the specification element in a "testable form", closer to the final corresponding Test Case, but unlike the latter, independently from the test harness specifics. In the ebXML Test Framework, a Test Requirement will be made of three parts:

- **Pre-condition** The pre-condition defines the context or situation under which this test item applies. It should help a reader understand in which case the corresponding specification requirement applies. In order to verify this Test Requirement, the test harness will attempt to create such a situation, or at the very least to identify when it occurs. If for some reason the pre-condition is not satisfied when doing testing, then it does not mean that the outcome of this test is negative – only that the situation in which it applies did not occur. In that case, the corresponding specification requirement could simply not be validated, and the subsequent Assertion will not be tested.

- **Assertion** The assertion actually defines the specification requirement, as usually qualified by a MUST or SHALL keyword. In the test harness, the verification of an assertion will be attempted only if the pre-condition is itself satisfied. When doing testing, if the assertion cannot be verified while the pre-condition was, then the outcome of this test item is negative.

- **Requirement Level** Qualifies the degree of requirement in the specification, as indicated by such keywords as RECOMMENDED, SHOULD, MUST, and MAY. Three levels can be distinguished: (1) "required" (MUST, SHALL), (2) "recommended" ([HIGHLY] RECOMMENDED, SHOULD), (3) "optional" (MAY, OPTIONAL). Any level lower than "required" qualifies a Test Requirement that is not mandatory for Conformance testing. Yet, lower requirement degrees may be critical to interoperability tests. The test requirement level can be override by explicit declaration in the Test Profile document, in case a lower or higher level is required.

5.2 The Test Requirements Document

The Test Requirements XML document provides metadata describing the Testing Requirements, their location in the specification, and their requirement type (REQUIRED, HIGHLY RECOMMENDED, RECOMMENDED, or OPTIONAL). A Test Profile driver file MUST validate against the ebXMLTestRequirements.xsd file found in Appendix B. The ebXML MS Conformance Test Requirements instance file can be found in Appendix E.

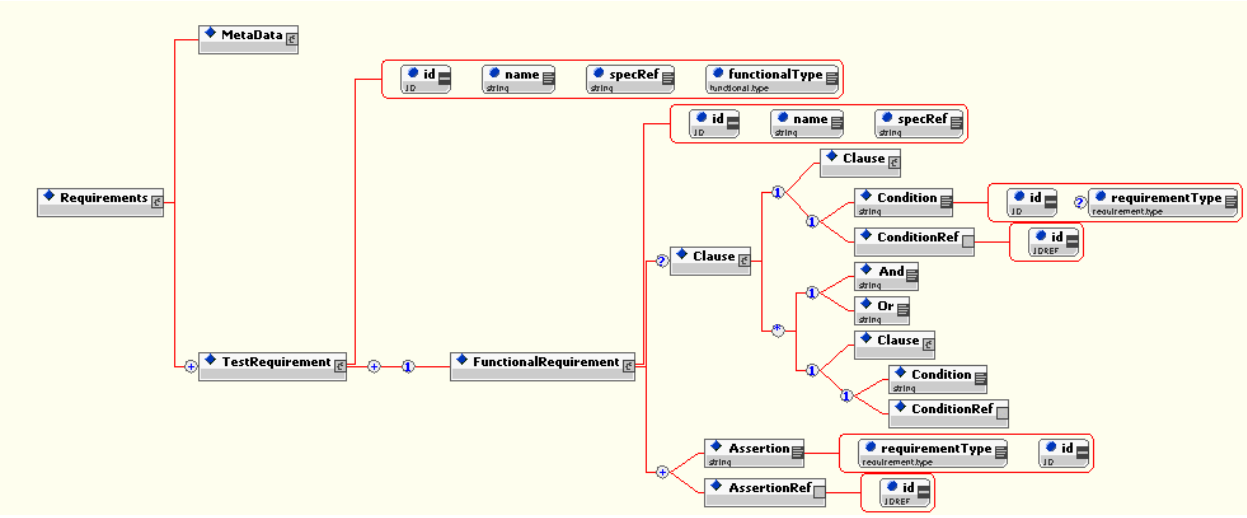


Figure 13 – Graphic representation of ebXMLTestRequirements.xsd schema

Definition of Content

Name	Description	Default Value From Test Driver	Required/Optional
Requirements	Container for all test requirements		Required
MetaData	Container for requirements metadata, including Description, Version, Maintainer, Location, Publish Date and Status		Required
Test Requirement	Container for all components of a single test requirement		Required
description	Description of requirement		Required
id	Unique identifier for each Test Requirement		Required
name	Name of test requirement		Required
specRef	Pointer to location in specification where requirement is found		Required
functionalType	Generic classification of function to be tested		Required
FunctionalRequirement	Sub-requirement for the main Test Requirement		Required
id	Unique ID for the sub-requirement		Required
name	Short descriptor of Functional Requirement		Required

specRef	Pointer to location in specification where sub-requirement is found		Required
Clause	Grouping element for Condition expression(s)		Optional
Condition	Textual description of test precondition		Required
ConditionRef	Reference (via id attribute) to existing Condition element already defined in the Test Requirements document		Required
And/Or	Union/Intersection operators for Conditions		Optional
Assertion	Axiom expressing expected behavior of an MSH implementation under conditions specified by any Clause		Required
AssertionRef	Reference (via id attribute) to existing Assertion element already defined in the Test Requirements document		Required
requirementType	Enumerated Assertion descriptor (REQUIRED, OPTIONAL…etc.)		Required

5.3 Specification Coverage

A Test Requirement is a formalized way to express a requirement of the target specification. The reference to the specification is included in each Test Requirement, and is made of one or more section numbers. There is no one-to-one mapping between sections of a specification document and the Test Requirement items listed in the test material for this specification:

- A specification section may map to several Test Requirements.
- A Test Requirement item may also cover (partially or not) more than one section or sub-section.

A Test Requirement item may then cover a subset of the requirements that are specified in a section.

For these reasons, it is important to determine to which degree the requirements of each section of a specification, are fully satisfied by the set of Test Requirements listed in the test suite document. This is done by establishing the Specification Coverage by the Test Requirements.

The Specification Coverage document is a separate document containing a list of all sections and subsections of a specification document, each annotated with:

- A coverage qualifier.
- A list of Test Requirements that map to this section.

The coverage qualifier may have values:

- **Full**: The requirements included in the specification document section are fully covered by the associated set of Test Requirements. This means that if each one of these Test Requirements is satisfied by an implementation, then the requirements of the corresponding document section are fulfilled. When the tests requirements are about conformance: The associated set of test requirement(s) are a clear indicator of conformance to the specification item, i.e. if a Candidate Implementation passes a Test Case that implements this test requirement(s) in a verifiable manner, there is a strong indication that it will behave similarly in all situations identified by the spec item.

- **None**: This section of the specification is not covered at all. Either there is no associated set of Test Requirements, or it is known that the test requirements cannot be tested even partially, at least with the Test Framework on which the test suite is to be implemented, and under the test conditions that are defined.

- **Partial**: The requirements included in this document section are only partially covered by the associated (set of) Test Requirement(s). This means that if each one of these Test Requirements is satisfied by an implementation, then it cannot be asserted that all the requirements of the corresponding document section are fulfilled: only a subset of all situations identified by the specification item are addressed. Reasons may be:

  - (1) The pre-condition(s) of the test requirement(s) ignores on purpose a subset of situations that cannot be reasonably tested under the Test Framework.
  - (2) The occurrence of situations that match the pre-condition of a Test Requirement is known to be under control of the implementation (e.g. implementation-dependent) or of external factors, and out of the control of the testbed. (See *contingent run-time coverage* definition, Section 7).

When the tests requirements are about conformance: The associated set of test requirement(s) are a weak indicator of conformance to the specification item. A negative test result will indicate non-conformance of the implementation.

5.4 Test Requirements Coverage (or Test Run-Time Coverage)

In a same way as Test Requirements may not be fully equivalent to the specification items they represent (see Specification Coverage, Section 5.3), the Test Cases that implement these Test Requirements may not fully verify them, for practical reasons.

Some Test Requirements may be difficult or impossible to verify in a satisfactory manner. The reason for this generally resides in an inability to satisfy the pre-condition. When processing a Test Case, the Test Harness will attempt to generate an operational context or situation that intends to satisfy the pre-condition, and that is supposed to be representative enough of real operational situations. The set of such real-world situations that is generally covered by the pre-condition of the Test Requirement is called the *test requirements (or test run-time) coverage* of this test Requirement. This happens in the following cases:

- **Partial run-time coverage**: It is in general impossible to generate all the situations that should verify a test. It is however expected that the small subset of run-time situations generated by the Test Harness, is representative enough of all real-world situations that are relevant to the pre-

condition. However, it is in some cases obvious that the Test Case definition (and its processing) will not generate a representative-enough (set of) situation(s). It could be that a significant subset of situations identified by the pre-condition of a Test Requirement cannot be practically set-up and verified. For example, this is the case when some combinations of events or of configurations of the implementation will not be tested due to the impracticality to address the combinatorial nature of their aggregation. Or, some time-related situations cannot be tested under expected time constraints.

- **Contingent run-time coverage**: It may happen that the test harness has no complete control in producing the situation that satisfies the pre-condition of a Test Requirement. This is the case for Test Requirements that only concern optional features that an implementation may or may not decide to exhibit, depending on factors under its own control and that are not understood or not easy to control by the test developers. An example is: " IF the implementation chooses to bundle together messages [e.g. under some stressed operation conditions left to the appreciation of this implementation] THEN the bundling must satisfy condition XYZ".

When a set of Test Cases is written for a particular set of Test Requirements, the degree of coverage of these Test Requirements by these Test Cases SHOULD be assessed. The Test Requirements coverage – not to be confused with the Specification Coverage - is represented by a list of the Test Requirements Ids, which associates with each Test Requirement:

- The Test Case (or set of Test Cases) that cover it,
- The coverage qualifier, which indicates the degree to which the Test Requirement is covered.

The coverage qualifier may have values:

- **Full**: the Test Requirement item is fully verified by the set of Test Cases.
- **Contingent**: The run-time coverage is contingent (see definition).
- **Partial**: the Test Requirement item is only partially verified by the associated set of Test Cases. The run-time coverage is partial (see definition).
- **None**: the Test Requirement item is not verified at all: there is no relevant Test Case.

6 Test Profiles

6.1 The Test Profile Document

The Test Profile document points to a subset of Test Requirements (in the Test Requirements document), that are relevant to the profile - either conformance, or interoperability profile - to be tested.

The document will drives the Test Harness by providing the Test Driver with a list of unique reference IDs of Test Requirements for a particular Test Profile. The Test Driver reads this document, and executes all Test Cases (located in the Test Suite document) that contain a reference to each of the test requirements. A Test Profile driver file MUST validate against the ebXMLTestProfile.xsd file found in Appendix A. A Test Profile example file can be found in section 10.2.

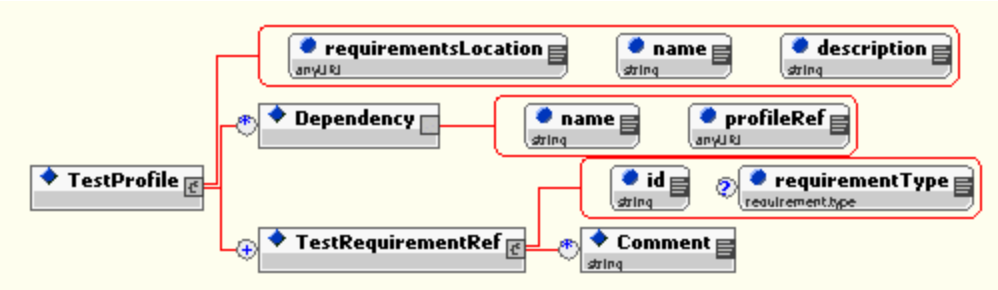


Figure 14 – Graphic representation of ebXMLTestProfile.xsd schema

Definition of Content

Name	Description	Default Value From Test Driver	Required/Optional
TestProfile	Container for all references to test requirements		Required
requirementsLocation	URI of test requirements XML file		Required
name	Name of profile		Required
description	Short description of profile		Required
Dependency	Prerequisite profile reference container		Optional
name	Name of the required prerequisite profile		Required
profileRef	Identifier of prerequisite profile to be loaded by Test Driver		Required

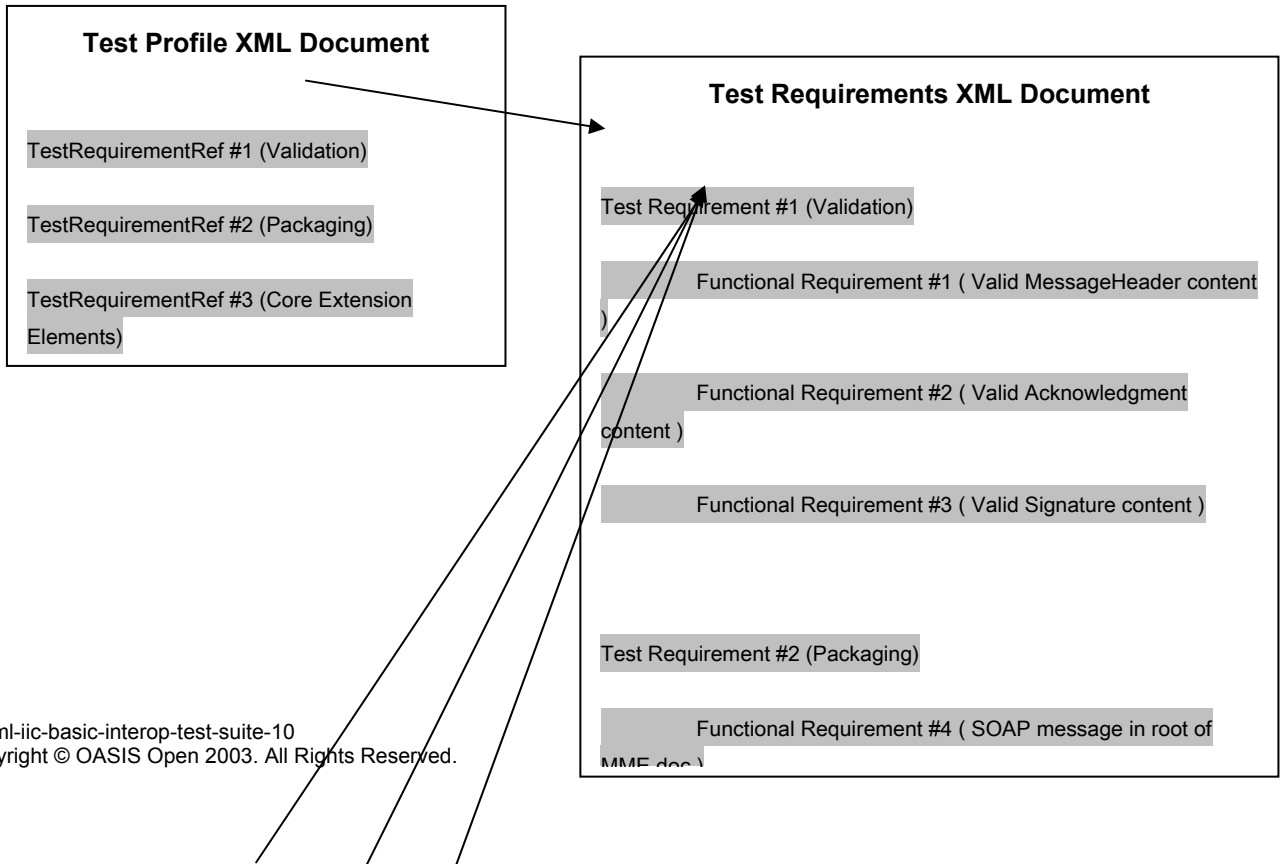
TestRequirementRef	Test Requirement reference		Required
id	Unique Identifier of Test Requirement, as defined in the Test Requirements document		Required
requirementType	Override existing requirement type with enumerated type of (REQUIRED, OPTIONAL, STRONGLY RECOMMENDED or RECOMMENDED)		Optional
Comment	Profile author's comment for a particular requirement		Optional

6.2 Relationships between Profiles, Requirements and Test Cases

Creation of a testing profile requires selection of a group of Test Requirement references that fulfill a particular testing profile. For example, to create a testing profile for a Core Profile would require the creation of an XML document referencing Test Requirements 1,2,3,4,5 and 8.

The Test Driver would read this list, and select (from the Test Requirements Document) the corresponding Test Requirements (and their "sub" Functional Requirements). The Test Driver then searches the Test Suite document to find all Test Cases that "point to" the selected Functional Requirements. If more than one Test Case is necessary to satisfactorily test a single Functional Requirement (as is the case for Functional Requirement #1) there may be more than one Test Case pointing to it. The Test Driver would then execute Test Cases #1, #2 and #3 in order to fully test an ebXML application against Functional Requirement #1.

The only test material outside of the three documents below that MAY require an external file reference from within a Test Case are large, or non-XML message Payloads



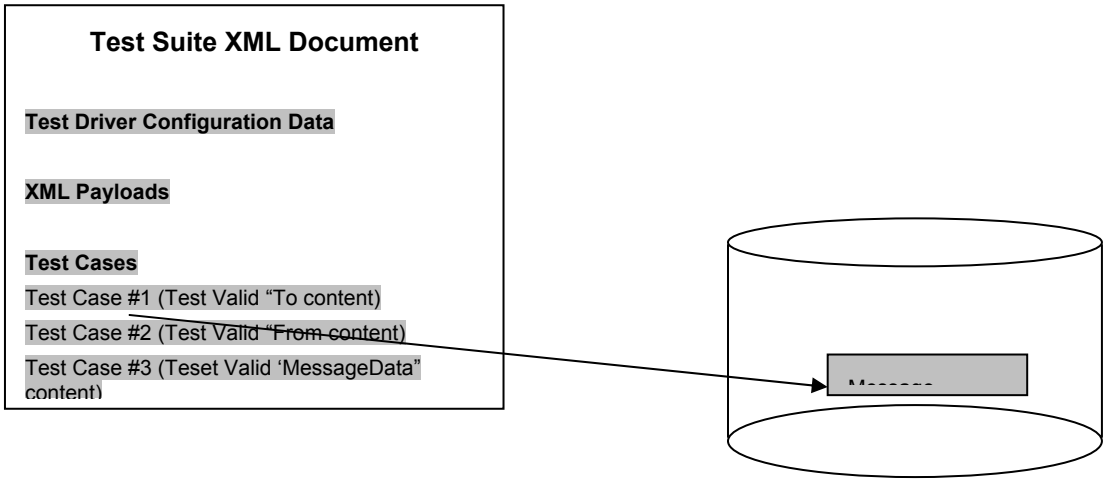


Figure 15 – Test Framework documents and their relationships

7 Test Cases

7.1 Structure of a Test Case

A Test Case is the translation of a Test Requirement (or a part of a Test Requirement), in an executable form, for a particular Test Harness. A Test Case includes the following information:

- Test Requirement reference.
- A Sequence of Test Steps.
- Condition(s) of success or of failure.

NOTE: The same Test Case may consolidate several Test Requirement items, i.e. a successful outcome of its execution will verify the associated set of Test Requirement items. This is usually the case when each of these Test Requirement items can make use of the same sequence of operations, varying only in the final test condition. When several Test Requirement items are covered by the same Test Case, the processing of the latter SHOULD produce separate verification reports.

Test Cases MUST evaluate to a Boolean value of “true/false” or (semantically) a “pass/fail”. The aggregated results of all Test Steps in a Test Case MUST evaluate to “true” for a Test Case result to be “pass”.

Prior to executing a Test Case, any configuration data necessary to modify the default configuration of the Test Driver MUST be included via a reference to a configurationGroupRef.

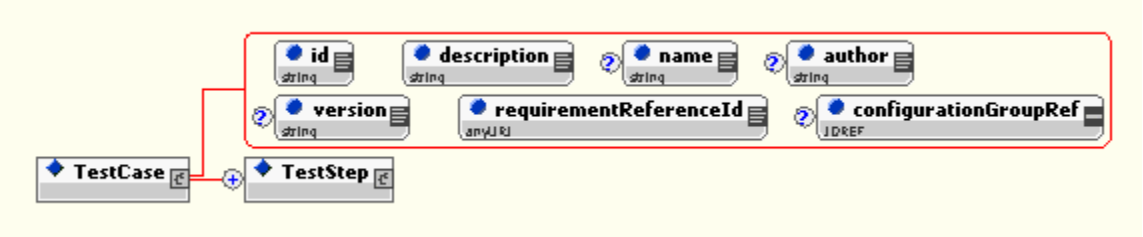


Figure 16 – Graphic representation of expanded view of the TestCase element

Definition of Content

Name	Description	Default Value From Test Driver	Required/Optional
id	Unique identifier for this Test Case		Required
description	Short description of TestCase		Optional
name	Short name for Test Case		Required
author	Name of person(s) creating the Test Case		Optional
version	Version number of Test Case		Optional
requirementReferenceId	Pointer to the unique ID of the FunctionalRequiremt		Required
configurationGroupRef	IDREF to reconfigure Test Driver using selected ConfigurationGroup		Optional
TestStep	Container for send, receive and message verification operations		Required

7.1.1 Test Steps

Test Steps are operations that MUST evaluate to a Boolean value of “true/false” or (semantically) a “pass/fail”. The aggregated results of all Test Steps in a Test Case MUST evaluate to “true” for a Test Case result to be “pass”.

Prior to executing a Test Step, any configuration data necessary to modify the default configuration of the Test Driver MUST be included via a reference to a configurationGroupRef.

7.1.2 Test Step Operations

Within a Test Step, the Test Driver may perform one of two main operations: message construction and transmission (PutMessage) or message retrieval and examination (GetMessage).

Associated with a Test Step are optional and required attributes. Altering the configuration of the Test Driver from its “bootstrap” or “base” configuration may be done through the addition of a configurationGroupRef attribute and its value.

Figure 17 – Graphic representation of expanded view of the TestStep element

Definition of Content

Name	Description	Default Value From Test Driver	Required/Optional
description	Short description of Test Step		Optional
configurationGroupRef	Reference to existing ConfigurationGroup to change current configuration for this Test Step		Optional
repeatTimes	Integer value indicating number of times this step should be repeated	1	Optional
testStepContext	Use CPAId, ConversationId, MessageId and RefToMessageId from previous step number indicated		Optional
stepDelay	Override the default delay between execution of this Test Step and the previous Test Step	Taken from current ConfigurationGroup value	Optional
PutMessage	Directive to construct and send an ebXML Message in its entirety (MIME, SOAP, and ebXML)		Required
GetMessage	Directive to retrieve messages from Message Store in their entirety		Required

7.1.3 The PutMessage Operation

The PutMessage Operation builds an ebXML message, along with its SOAP and MIME containers. A minimal Message Declaration is required to create a message, with default values for MIME headers and ebXML message content provided by the Test Driver. A Message Declaration described in a PutMessage Operation MUST validate against the ebXMLTestSuite.xsd schema in Appendix C.

The message components that can be created and modified by PutMessage include:

- **MIME header data:** MIME headers MUST be created or modified using the declarative syntax described in the ebXMLTestSuite.xsd schema in Appendix C. Default message MIME header data is illustrated in the message envelope template in section 7.1.6. How the MIME headers are

actually constructed is implementation dependent. Test Drivers operating in “service” mode MAY ignore the MIME portion of a Message Declaration, since message MIME manipulation may be unavailable at the application level interface used for a particular ebXML MSH implementation. Test drivers in “connection” mode MUST properly interpret the MIME portion of a Message Declaration and generate the appropriate MIME header information.

- **SOAP header and body data:** SOAP message content MUST be created or modified using the declarative syntax described in the ebXMLTestSuite.xsd schema described in Appendix C. Default message SOAP content is illustrated in the message envelope template in section 7.1.6. How the actual SOAP message is constructed is implementation dependent. Test Drivers operating in “service” mode MAY ignore the SOAP portion of a MessageDeclaration, since message SOAP manipulation may be unavailable at the application level interface used for an MSH implementation. Test drivers in “connection” mode MUST properly interpret the SOAP portion of a Message Declaration and generate the appropriate SOAP header information.

- **ebXML Message data:** ebXML message content MUST be created or modified using the declarative syntax described in the ebXMLTestSuite.xsd schema described in Appendix C. How the actual ebXML message is constructed is implementation dependent. Test drivers operating in both “service” and “connection” modes MUST properly interpret the ebXML portion of a Message Declaration, and generate the appropriate ebXML content.

- **ebXML payload data:** ebXML message payload content, when there is a payload, MUST be created or modified using the syntax described in the ebXMLTestSuite.xsd schema described in Appendix C. ebXML payloads are created through file or through an XML ID reference inclusion into a message, and MAY be modified through any implementation-specific XML syntax.

In addition to creating message content, the PutMessage operation can also execute sub-operations to create payloads, digitally sign any portion of the message and set global Test Case parameters that can be used by other Test Steps in the current Test Case in their XPath evaluation of message content.

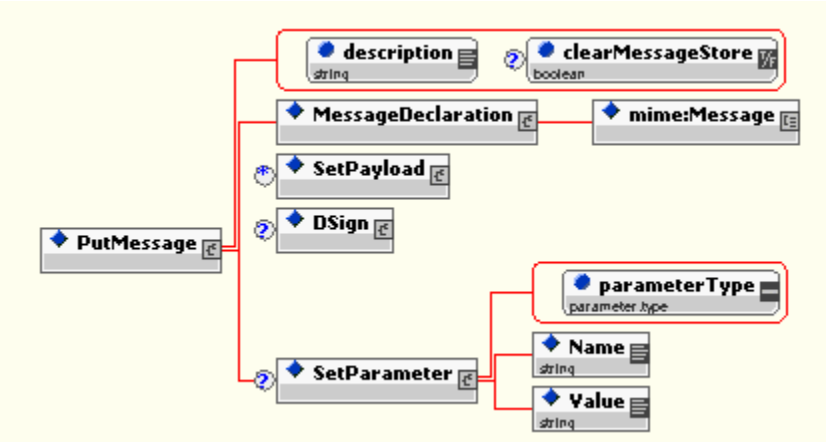


Figure 18 – Graphic representation of expanded view of the PutMessage element

Definition of Content

Name	Description	Default Value From Test Driver	Required/Optional
description	Metadata describing the nature of the PutMessage operation		Required
clearMessageStore	Boolean attribute directive to purge the Test Driver Message Store of all previous received messages for this particular Test Case	false	Optional
MessageDeclaration	Container for XML content to construct the message		Required
SetPayload	Container element for Test Driver directives to create MIME attachments (or Payloads) to message		Optional
DSign	Container element for XML Digital Signature declaration(s) for this message		Optional
SetParameter	Container for user-defined parameter to be made available to other Test Steps (source can be this message's content retrieved via XPath statement or simply a user-supplied string value)		Optional
parameterType	Choice of Xpath or string		Required
Name	Name of new parameter		Required
Value	String value or XPath expression pointing to desired element/attribute value in message or simply a user-supplied string value		Required

7.1.4 The Message Declaration

The MessageDeclaration element is a container element for XML content describing the construction of MIME, SOAP and ebXML portions of a message. The XML content necessary to describe a basic message is minimal, with default parameter values supplied by the Test Driver for most message content. If the test developer wishes to "override" the default element and attribute values, they may do so by explicitly declaring those values in the XML markup.

Default values for element and attribute content may come from two sources. Either the Test Driver/MSH generates that value, (such as for message Timestamp), or the value is declared in the ConfigurationGroup parameters described in section 4.2.1. For example, a Test Suite ConfigurationGroup element content may be:

```
<ebTest:ConfigurationGroup ebTest:id="cpa_basic">
  <ebTest:CPAId>cpa_basic</ebTest:CPAId>
  <ebTest:Mode>connect</ebTest:Mode>
  <ebTest:SenderParty>urn:oasis:iic:testdriver</ebTest:SenderParty>
```

```
  <ebTest:ReceiverParty>urn:oasis:iic:testservice</ebTest:ReceiverParty>
  <ebTest:Service>urn:ebXML:iic:test.</ebTest:Service>
  <ebTest:Action>Dummy</ebTest:Action>
</ebTest:ConfigurationGroup>
```

A Test Driver could then read the MessageDeclaration (below) authored by the test writer and build the message, inserting element and attribute content wherever it knows default content should be, and declaring, or overriding default values where they are explicitly defined in the Message Declaration.

```
<ebTest:MessageDeclaration>
 <mime:Message>
  <mime:MessageContainer>
   <soap:Envelope>
    <soap:Header>
     <eb:MessageHeader/>
    </soap:Header>
    <soap:Body />
   </soap:Envelope>
  </mime:MessageContainer>
 </mime:Message>
</ebTest:MessageDeclaration>
```

For illustrative purposes, the resulting message can be represented by the example message below. The Test Driver, after parsing the simple Message Declaration above, would generate the following MIME message with enclosed SOAP/ebXML content.

```
Content-Type: multipart/related; type="text/xml"; boundary="boundaryText";
start=messagepackage@oasis.org

--boundaryText

Content-ID: <messagepackage@oasis.org>
Content-Type: text/xml;  charset="UTF-8"

<soap:Envelope    xmlns:xlink="http://www.w3.org/1999/xlink"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:eb="http://www.oasis-open.org/committees/ebxml-msg/schema/msg-header-
2_0.xsd"          xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/

http://www.oasis-open.org/committees/ebxml-msg/schema/envelope.xsd

http://www.oasis-open.org/committees/ebxml-msg/schema/msg-header-2 0.xsd

http://www.oasis-open.org/committees/ebxml-msg/schema/msg-header-2_0.xsd">
<soap:Header>
   <eb:MessageHeader soap:mustUnderstand="1" eb:version="2.0">
```

```
            <eb:From>
                    <eb:PartyId>urn:oasis:iic:testdriver<</eb:PartyId>
            </eb:From>
            <eb:To>
                    <eb:PartyId>urn:oasis:iic:testservice</eb:PartyId>
            </eb:To>
            <eb:CPAId> urn:config:cpa_basic</eb:CPAId>
            <eb:ConversationId> 987654321</eb:ConversationId>
            <eb:Service>urn:ebXML:iic:test</eb:Service>
            <eb:Action>Dummy</eb:Action>
            <eb:MessageData>
                    <eb:MessageId>0123456789<</eb:MessageId>
                    <eb:Timestamp>2000-07-25T12:19:05</eb:Timestamp>  MessageData>
    </eb:MessageHeader>
</soap:Header>
</soap:Envelope>

--boundaryText
```

Certain XML message values (illustrated in red above) are supplied by the Test Driver, and are also made available to subsequent Test Step operations through global XPath parameter names. These parameters define the Test Step Context:

- $CPAId
- $ConversationId
- $MessageId
- $RefToMessageId
- $Service
- $Action
- $SenderParty
- $ReceiverParty

Some of these values dynamically change with each PutMessage operation. These include MessageId. Other's change with each Test Case ($ConversationId). Still other parameters remain "constant" within the scope of the current ConfigurationGroup definition ($CPAId, $Service, $Action, $SenderParty and $ReceiverParty). ALL can be "overridden" through explicit declaration in the Message Declaration.

The ebXMLTestSuite.xsd schema in Appendix C defines the format for element and attribute content declaration. However, the schema alone DOES NOT define default element content, since this is beyond the capability of schemas. Therefore, Test Driver implementers MUST consult the "Definition of Content" tables for this section of the specification to determine what default XML content must be generated by the Test Driver or MSH to create a valid ebXML message.

The following sections description of how a Test Driver or MSH MUST interpret the MessageDeclaration content in order to be conformant to this specification.

7.1.4.1 Interpreting the MIME portion of the Message Declaration

The XML syntax used by the Test Driver to construct the MIME message content consists of the declaration of a main MIME container for the entire message, followed by a MIME container for the SOAP message envelope. Default values for MIME headers MAY be “overridden” by explicit declaration of their values in the MessageDeclaration content; otherwise, default values are used by the Test Driver to construct the MIME headers.

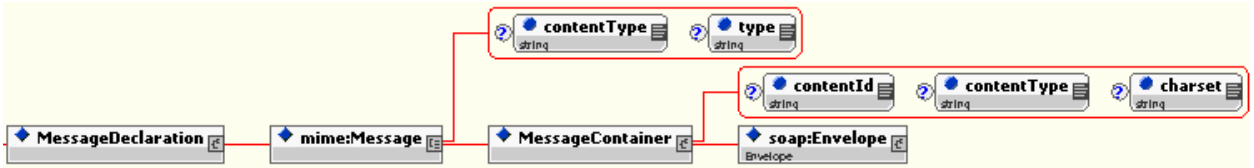


Figure 19 – Graphic representation of expanded view of the MessageDeclaration element

Definition of Content

Name	Declaration Description	Default Value From Test Driver	Required/Optional
mime:Message	Generate container for MIME, SOAP and ebXML message content		Required
contentType	Generate a MIME message ‘Content-Type’ header	multipart/related	Optional
type	Generate a MIME message ‘type’ header	text/xml	Optional
MessageContainer	Generate a MIME container in message		Required
contentId	Generate a ‘Content-ID’ MIME header for the container	messagepackage @oasis.org	Optional
contentType	Generate a MIME message package ‘Content-Type’ header	text/xml	Optional
charset	Generate a MIME message package character set	UTF-8	Optional
soap:Envelope	Generates a MIME container for SOAP message		Required

An Example of Minimal MIME Declaration Content

The following XML represents all the information necessary to permit a Test Driver to construct a MIME message that may contain a SOAP envelope in its first MIME container. The XML document below validates against the ebXMLTestSuite.xsd schema in Appendix C.

```
<mime:Message xmlns:mime="http://www.oasis-open.org/tc/ebxml-iic/testing/mime">
  <mime:MessageContainer/>
```

```
</mime:Message>
```

7.1.4.2 Interpreting the SOAP portion of the Message Declaration

The XML syntax interpreted by the Test Driver to construct the SOAP message content consists of the declaration of a SOAP Envelope element, which in turn is a container for the SOAP Header, Body and non-SOAP XML content. Construction of the SOAP Header and Body content is simple for the Test Driver, requiring only the creation of the two container elements with their namespace properly declared, and valid according to the [SOAP]. The SOAP Body element, or any “wildcard” XML content is only constructed by the Test Driver if it is explicitly declared in the content.

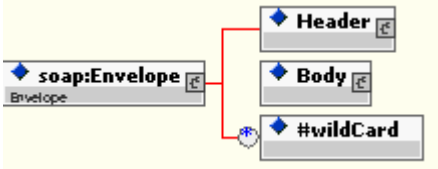


Figure 20 – Graphic representation of expanded view of the soap:Envelope element declaration

Definition of Content

Name	Declaration Description	Default Value From Test Driver	Required/Optional
soap:Envelope	Generate container element with its proper namespace for SOAP Header and Body elements and their content		Required
soap:Header	Generate SOAP Header extension element		Required
soap:Body	Modify the default Body element	Element is auto-generated by Test Driver at run time	Optional
#wildCard	Generate “inline” wildcard content inside SOAP Envelope		Optional

An Example of Minimal SOAP Declaration Content

The following XML represents all the information necessary to permit a Test Driver to construct a minimal SOAP message.

```
<soap:Envelope>
  <soap:Header/>
 </soap:Envelope>
```

7.1.4.3 Interpreting the SOAP Header Extension Element Declaration

The declarative syntax interpreted by the Test Driver to construct the ebXML Header extension message content consists of the declaration of a SOAP Header element, which in turn is a container for the ebXML Header extension elements and their content. The only extension element that is required in the container is the eb:MessageHeader element, which directs the Test Driver to construct an ebXML MessageHeader element, along with its proper namespace declaration, as defined in [EBMS]. The Test Driver does not construct any other Header extension elements unless they are explicitly declared as content in the SOAP Header Declaration.

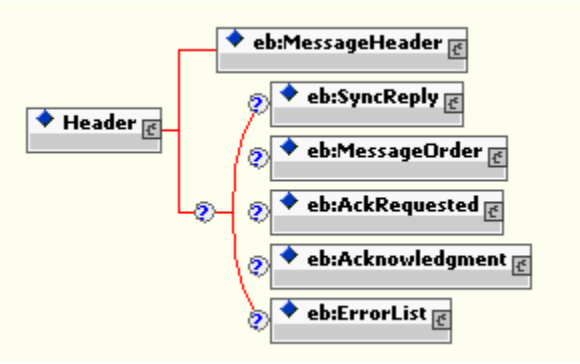


Figure 21 – Graphic representation of expanded view of the soap:Header element declaration

Definition of Content

Name	Declaration Description	Default Value From Test Driver	Required/Optional
Header	SOAP Header declaration and container for ebXML ebXML Header Extension Element declarations		Required
eb:MessageHeader	Create an ebXML MessageHeader element with namespace declaration		Required

eb:ErrorList	Create an ebXML ErrorList element		Optional
eb:SyncReply	Create an ebXML SyncReply element		Optional
eb:MessageOrder	Create an ebXML MessageOrder element		Optional
eb:AckRequested	Create an ebXML AckRequested element		Optional
eb:Acknowledgment	Create an ebXML Acknowledgment element		Optional

7.1.4.4 Interpreting the ebXML MessageHeader Element Declaration

The XML syntax interpreted by the Test Driver to construct the ebXML MessageHeader extension content consists of the declaration of a MessageHeader element, and a required declaration of CPAId and Action elements within it. This is the ”minimum” declaration aTest Driver needs to generate an ebXML Message Header. All other required content, as defined in the schema in the ebXML MS v2.0 Specification, is provided by the Test Driver through either default parameters defined in the ebXMLTestSuite.xsd schema in Appendix C, or directly generated by the Test Driver (e.g. to generate necessary message container elements) or by explicit declaration of content in the Message Declaration. The figure below illustrates the schema for an ebXML Message Header declaration to be interpreted by the Test Driver.

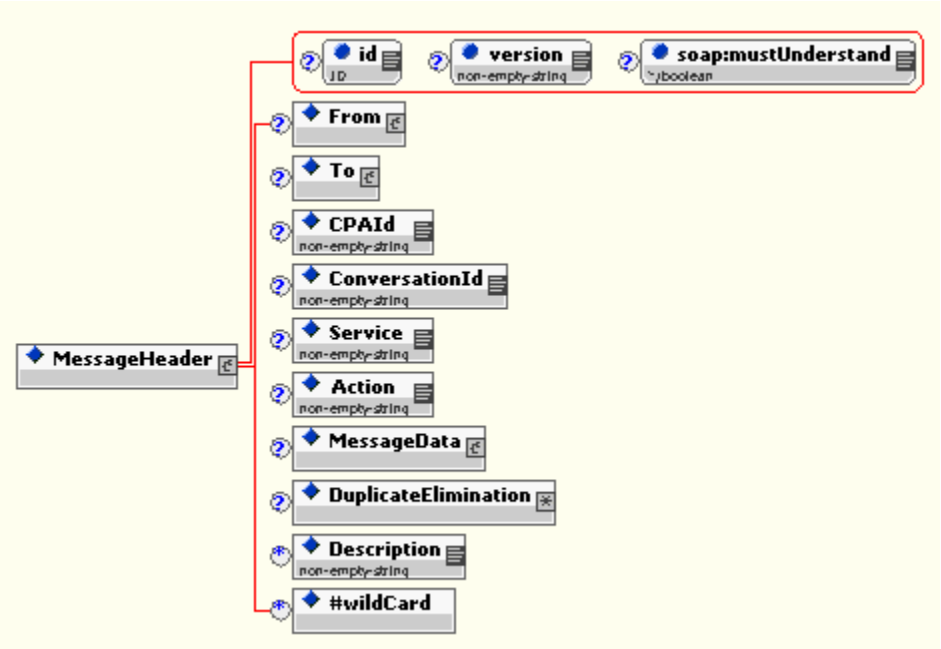


Figure 22 – Graphic representation of expanded view of the ebXML MessageHeader element declaration

Definition of Content

Name	Declaration Description	Default Value From Test Driver	Required/Optional
eb:MessageHeader	Generate MessageHeader element and all of its default element/attribute content		Required
id	Generate attribute with declared value		Optional
version	Modify default attribute value	2.0	Optional
soap:mustUnderstand	Modify default attribute value	true	Optional
From	Modify default From message element generated by Test Driver	Generated by Test Driver/MSH at run time	Optional
PartyId	Replace default element value with new value	Generated by Test Driver/MSH at run time, using config value	Required
type	Generate a type attribute with value		Optional
Role	Generates a Role element with its value		Optional
To	Modify default To message element generated by Test Driver	Generated by Test Driver at run time	Optional
PartyId	Replace default element value with new value	Generated by Test Driver/MSH at run time, using config value	Required
type	Generate type attribute with value		Optional
Role	Generates a Role element with its value		Optional
CPAId	Generate element with its value	Generated by Test Driver/MSH at run time, using config value	Optional
ConversationId	Modify default value provided by Test Driver	Generated by Test Driver at run time	Optional
Service	Modify default value generated by Test Driver	Generated by Test Driver/MSH at run time, using config value	Optional
Action	Replace default value with	Generated by Test	

	specified Action name	Driver/MSH at run time, using config value	Optional
MessageData	Modify default container generated by Test Driver	Generated by Test Driverat run time	Optional
MessageId	Modify default value generated by Test Driver	Generated by Test Driver at run time	Optional
Timestamp	Modify default value generated by Test Driver	Generated by Test Driver at run time	Optional
RefToMessageId	Generate element and its value		Optional
TimeToLive	Generate element and its value	Generated by Test Driver at run time	Optional
DuplicateElimination	Generate element		Optional
Description	Generate element with value		Optional
#wildcard	Generate content inline		Optional

An Example of a Minimal ebXML MessageHeader Content Declaration

The following XML represents all the information necessary to permit a Test Driver to construct an ebXML MessageHeader element with all necessary content to validate against the ebXML MS V2.0 schema. All declared content must validate the ebXMLTestSuite.xsd schema in Appendix C.

```
<eb:MessageHeader/>
```

7.1.4.5 Interpreting the ebXML ErrorList Element Declaration

The XML syntax interpreted by the Test Driver to construct the ebXML ErrorList extension content consists of the declaration of an ErrorList element, and a required declaration of one or more Error elements within it. All required content, as defined in the schema in the ebXML MS V2.0 Specification, is provided through either default parameters defined in the ebXMLTestSuite.xsd schema and included by the Test Driver, or by explicit declaration.

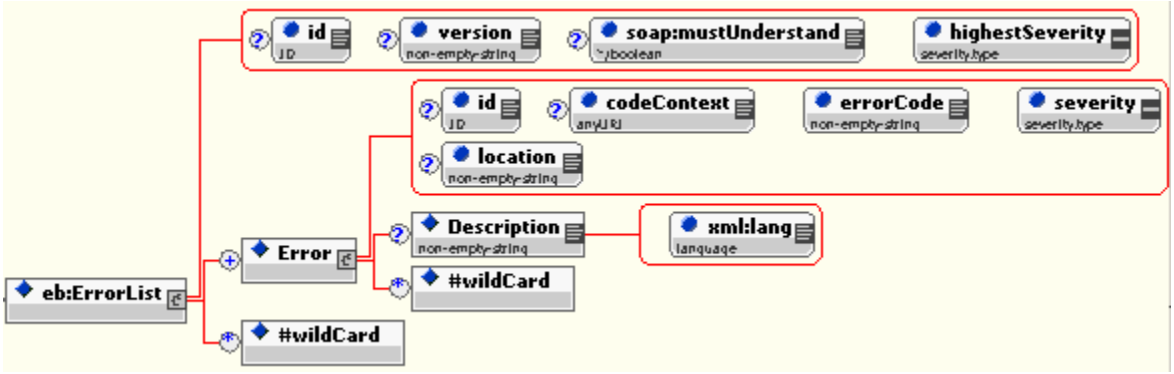


Figure 23 – Graphic representation of expanded view of the ebXML ErrorList element declaration

Definition of Content

Name	Declaration Description	Default Value From Test Driver	Required/Optional
eb:ErrorList	Generate container element		Optional
id	Generate attribute and its value		Optional
version	Modify default value	2.0	Optional
soap:mustUnderstand	Modify default value	true	Optional
highestSeverity	Generate required attribute and value		Required
Error	Generate new Error container		Required
id	Generate attribute with declared value		Optional
codeContext	Generate element with declared value		Optional
errorCode	Generate required attribute and value		Required
severity	Generate required attribute and value		Required
location	Generate attribute with declared value		Optional
Description	Generate element with declared value		Optional
#wildCard	Generate content “inline” into message		Optional

An Example of a Minimal ebXML ErrorList Content Declaration

The following XML represents all the information necessary to permit a Test Driver to construct an ebXML ErrorList element with all necessary content to validate against the ebXML MS v2.0 schema. All required content not visible in the example would be generated by the Test Driver.

```
<eb:ErrorList eb:highestSeverity=Error">
   <eb:Error eb:errorCode="Inconsistent" eb:severity="Error"/>
</eb:ErrorList>
```

7.1.4.6 Interpreting the ebXML SyncReply Element Declaration

The XML syntax interpreted by the Test Driver to construct the ebXML SyncReply extension content consists of the declaration of a SyncReply element. All required content, as defined in the schema in [EBMS], is provided through either default parameters provided by the Test Driver or through explicit declaration.

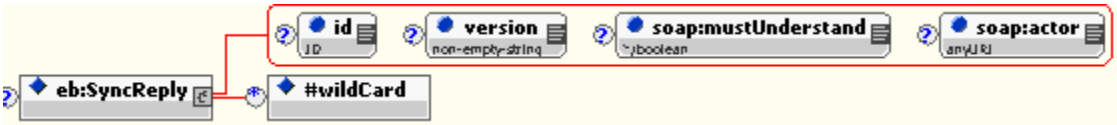


Figure 24 – Graphic representation of expanded view of the ebXML SyncReply element declaration

Definition of Content

Name	Declaration Description	Default Value From Test Driver	Required/Optional
eb:SyncReply	Generate container element and all default content		Optional
id	Generate attribute and its value		Optional
version	Modify default attribute value	2.0	Optional
soap:mustUnderstand	Modify default attribute value	true	Optional
soap:actor	Modify default attribute value	http://schemas.xmlsoap.org/soap/actor/next	Optional
#wildCard	Generate content “inline”		Optional

An Example of a Minimal ebXML SyncReply Content Declaration

The following XML represents all the information necessary to permit a Test Driver to construct an ebXML AckRequested element with all necessary content to validate against the [EBMS] schema schema.

```
<eb:SyncReply/>
```

7.1.4.7 Interpreting the ebXML AckRequested Element Declaration

The XML syntax interpreted by the Test Driver to construct the ebXML AckRequested extension content consists of the declaration of an AckRequested element. All required content as defined in the [EBMS] schema, is provided by the Test Driver or by explicit declaration.

Figure 25 – Graphic representation of expanded view of the ebXML AckRequested element declaration

Definition of Content

Name	Declaration Description	Default Value From Test Driver	Required/Optional
eb:AckRequested	Generate container element and all default content		Optional
id	Generate attribute and its value		Optional
version	Modify default value	2.0	Optional
soap:mustUnderstand	Modify default value	true	Optional
soap:actor	Modify default attribute value with new value	urn:oasis:names:tc:ebxml-msg:actor:toPartyMSH	Optional
signed	Modify default attribute value	false	Optional
#wildCard	Generate content “inline”		Optional

An Example of a Minimal ebXML AckRequested Content Declaration

The following XML represents all the information necessary to permit a Test Driver to construct an ebXML AckRequested element with all necessary content to validate against the [EBMS] schema.

```
<eb:AckRequested/>
```

7.1.4.8 Interpreting the ebXML Acknowledgment Element Declaration

The XML syntax interpreted by the Test Driver to construct the ebXML Acknowledgment extension content consists of the declaration of an Acknowledgment element. All required content, as defined in the [EBMS] schema, is provided by the Test Driver or through explicit declaration.

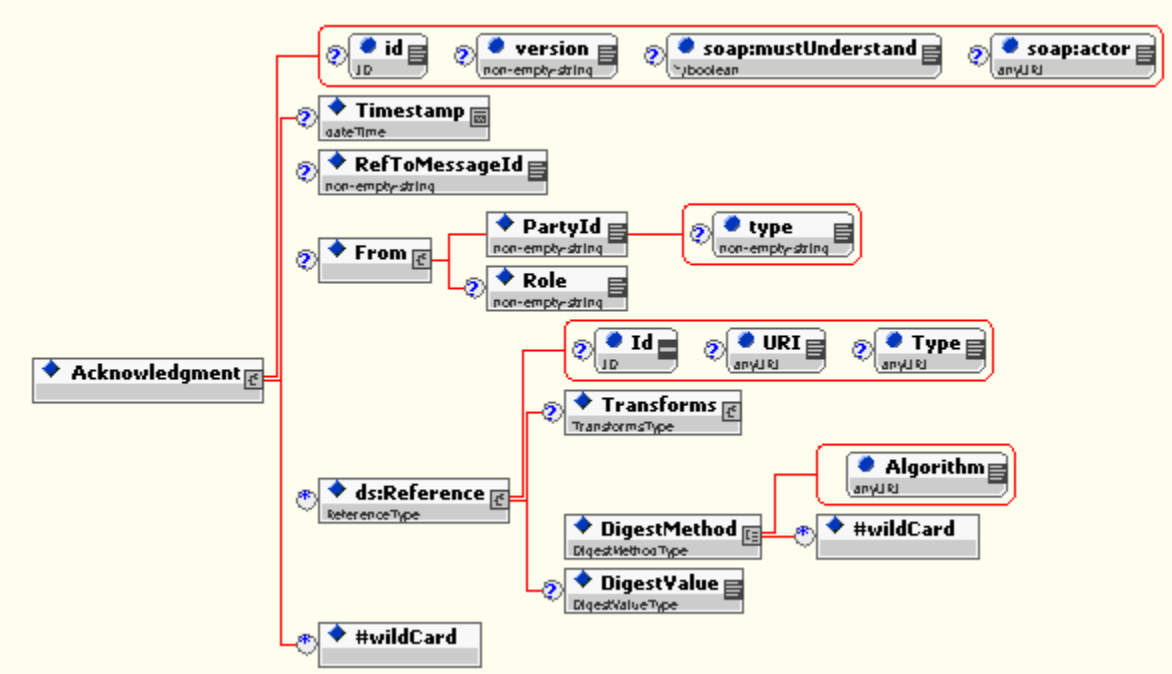


Figure 26 – Graphic representation of expanded view of the ebXML Acknowledgment element declaration

Definition of Content

Name	Declaration Description	Default Value From Test Driver	Required/Optional

eb:Acknowledgment	Generate container element and all default content		Optional
id	Generate attribute and its value		Optional
version	Modify default attribute value	2.0	Optional
soap:mustUnderstand	Modify default attribute value	true	Optional
soap:actor	Modify default attribute value	urn:oasis:names:tc:ebxml-msg:actor:toPartyMSH	Optional
Timestamp	Modify default element value	Generated by Test Driver at run time	Optional
RefToMessageId	Modify default element value	Generated by Test Driver at run time	Optional
From	Modify default container	Generated by Test Driver at run time	Optional
PartyId	Modify default value	urn:ebxml:iic:testdriver	Required
type	Generate type attribute with value		Optional
Role	Generates a Role element with its value		Optional
ds:Reference	Generate container element and all default content		Optional
Id	Generate attribute and its value		Optional
URI	Modify default attribute value	“”	Required
type	Generate attribute and its value		Optional
Transforms	Generate container relement		Optional
Transform	Generate element with its value		Optional
Algorithm	Modify default attribute value	http://www.w3.org/TR/2001/REC-xml-c14n-20010315	Required
#wildCard	Generate content “inline”		Optional
XPath	Generate element with its value		Optional
DigestMethod	Generate element with its value		Required
Algorithm	Modify default attribute value	Generated by Test Driver at run time, based upon CPA	Required
#wildCard	Generate content “inline”		Optional

DigestValue	Generate element with its value	Computed by Test Driver at run time	Required
#wildCard	Generate content “inline”		Optional

An Example of a Minimal “unsigned” ebXML Acknowledgment Content Declaration

The following XML represents the minimum information necessary to permit a Test Driver to construct an ebXML Acknowledgment element.

```
<eb:Acknowledgment/>
```

7.1.4.9 Interpreting the ebXML MessageOrder Element Declaration

The XML syntax interpreted by the Test Driver to construct the ebXML MessageOrder extension content consists of the declaration of a MessageOrder element. All required content, as defined in the [EBMS] schema, is provided by the Test Driver or through explicit declaration.

Figure 27 – Graphic representation of expanded view of the ebXML MessageOrder element declaration

Definition of Content

Name	Declaration Description	Default Value From Test Driver	Required/Optional
eb:MessageOrder	Generate container element and all default content		Optional
id	Generate attribute and its value		Optional
version	Modify default attribute value	2.0	Optional
soap:mustUnderstand	Modify default attribute value	true	Optional
SequenceNumber	Generate element with declared value		Required

status	Generate attribute with declared value		Optional
#wildCard	Generate content “inline”		Optional

An Example of a Minimal ebXML MessageOrder Content Declaration

The following XML represents all the information necessary to permit a Test Driver to construct an ebXML MessageOrder element.

```
<eb:MessageOrder>
<eb:SequenceNumber>1</eb:SequenceNumber>
</eb:MessageOrder>
```

7.1.4.10 Interpreting the SOAP Body Extension Element Declaration

The XML syntax used by the Test Driver to construct the ebXML Body extension message content consists of the declaration of a SOAP Body element, which in turn is a container for the ebXML Manifest, StatusRequest or StatusResponse elements.

The Test Driver does not construct any of these SOAP Body extension elements unless they are explicitly declared as content in the SOAP Body Declaration.

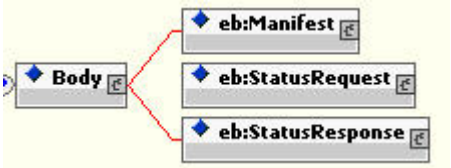


Figure 28 – Graphic representation of expanded view of the soap:Body element declaration

7.1.4.11 Interpreting the ebXML Manifest Element Declaration

The XML syntax interpreted by the Test Driver to construct the ebXML Manifest extension content consists of the declaration of a Manifest element. All required content, as defined in the [EBMS] schema, is provided by the Test Driver or through explicit declaration

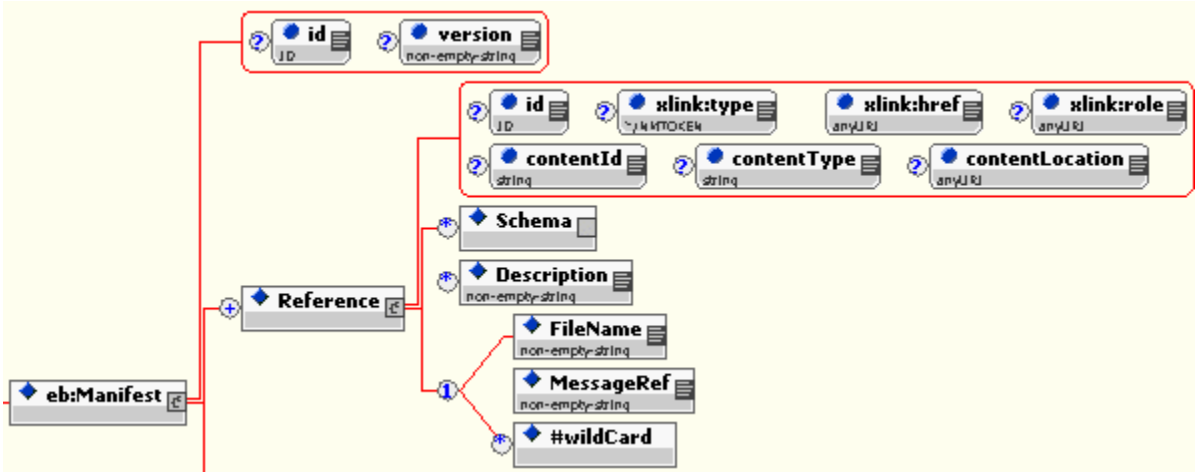


Figure 29 – Graphic representation of expanded view of the ebXML Manifest element declaration

Definition of Content

Name	Declaration Description	Default Value From Test Driver	Required/Optional
eb:Manifest	Generate container element and all default content		Optional
id	Generate attribute and its value		Optional
version	Modify default attribute value	2.0	Optional
id	Modify default attribute value	true	Optional
xlink:type	Generate element with declared value		Optional
xlink:href	Generate attribute with declared value		Required
xlink:role	Generate attribute with declared value		Optional
contentId	Modify the Content-ID MIME header of the payload		Optional
contentType	Set the the Content-Type MIME header of the payload		Optional
contentLocation	Set the the Content-Location MIME header of the payload		Optional
Schema	Generate schema container element		Optional
location	Generate URI attribute and value of schema location		Required
version	Generate schema version attribute and value		Optional
Description	Generate description element and value		Optional

xml:lang	Generate description language attribute and value		Required
PayloadLocation	Load specified file as a MIME attachment to message		Required
MessageRef	Load designated XML document via IDREF as a MIME attachment to message		Required
PayloadDeclaration	“Inline” the XML content of this element as a MIME message attachment		Required

An Example of a Minimal ebXML Manifest Content Declaration

The following XML represents the minimum information necessary to permit a Test Driver to construct an ebXML Manifest element with all necessary content to validate against the ebXML MS v2.0 schema.

```
<eb:Manifest>
<eb:Reference xlink:href=”cid:payload 1”/>
</eb:Manifest>
```

7.1.4.12 Interpreting the ebXML StatusRequest Element Declaration

The XML syntax interpreted by the Test Driver to construct the ebXML StatusRequest extension content consists of the declaration of a StatusRequest element. All required content, as defined in the [EBMX] schema. All required content, as defined in the [EBMS] schema, is provided by the Test Driver or through explicit declaration

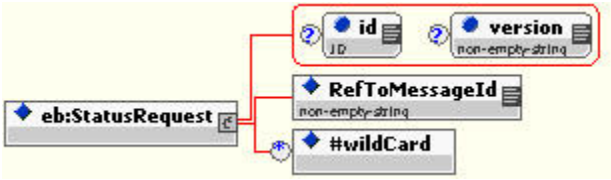


Figure 30 – Graphic representation of expanded view of the ebXML StatusRequest element declaration

Definition of Content

Name	Declaration Description	Default Value From Test Driver	Required/Optional
eb:StatusRequest	Generate container element and all default content		Optional
id	Generate attribute and its value		Optional

version	Modify default value	2.0	Optional
RefToMessageId	Generate element and its value		Required
#wildCard	Generate content “inline”		Optional

An Example of a Minimal ebXML StatusRequest Content Declaration

The following XML represents all the minimum information necessary to permit a Test Driver to construct an ebXML StatusRequest element with all necessary content to validate against the [EBMS] schema.

```
<eb:StatusRequest>
<eb:RefToMessageId>20001209-133003-28571@example.com</eb:RefToMessageId>
</eb:StatusRequest>
```

7.1.4.13 Interpreting the ebXML StatusResponse Element Declaration

The XML syntax used by the Test Driver to construct the ebXML StatusResponse extension content consists of the declaration of a StatusResponse element with required and optional element/attribute content.

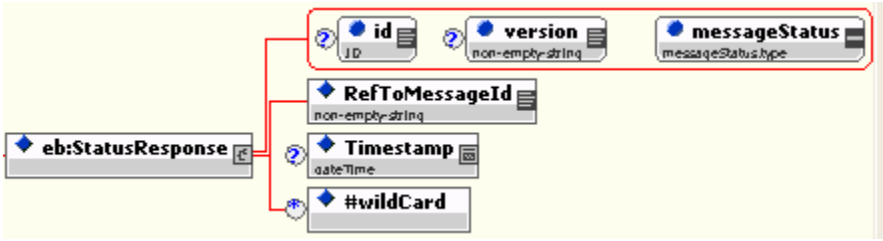


Figure 31 – Graphic representation of expanded view of the ebXML StatusResponse element declaration

Definition of Content

Name	Declaration Description	Default Value From Test Driver	Required/Optional
eb:StatusResponse	Generate container element and all default content		Optional
id	Generate attribute and its value		Optional
version	Modify default attribute value	2.0	Optional
messageStatus	Generate attribute and its		Optional

	value		
RefToMessageId	Generate element and its value		Required
Timestamp	Modify default value	Generated by Test Driver at run time	Optional
#wildCard	Generate content “inline”		Optional

An Example of a Minimal ebXML StatusResponse Content Declaration

The following XML represents all the information necessary to permit a Test Driver to construct an ebXML StatusResponse element with all necessary content to validate against the [EBMX] schema.

```
<eb:StatusResponse messageStatus="Processed"/>
```

7.1.5 The SetPayload Operation

The SetPayload Operation is a sub-operation for the PutMessage Operation. It provides the Test Driver with the necessary information to append a message payload. Payloads can be provided to the driver through a file name reference, an in-memory message document reference, or can be constructed “on-the-fly” through any declarative syntax specific to an ebXML application.

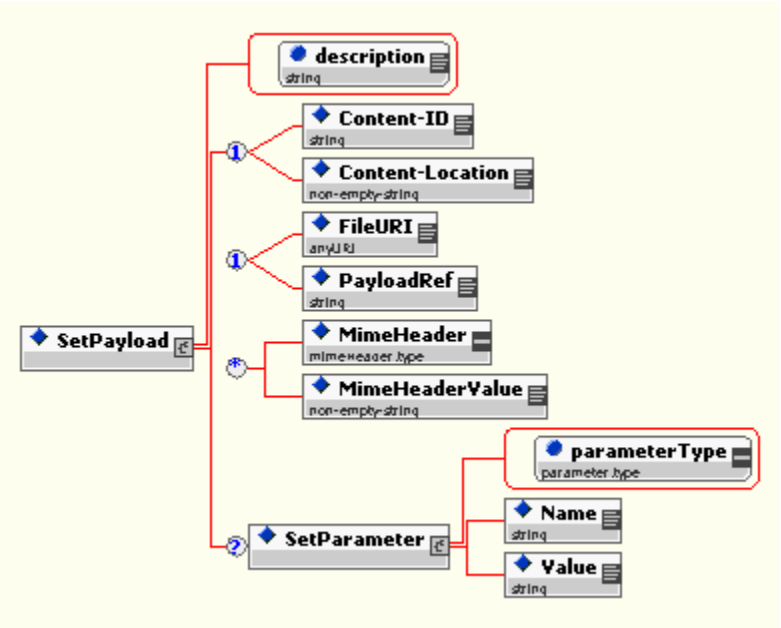


Figure 32 – Graphic representation of expanded view of the SetPayload element

Definition of Content

Name	Description	Default Value	Required/Optional
description	Metadata describing the nature of the SetPayload operation		Required
Content-ID	Set the Content-Id MIME header of the payload		Required
Content-Location	Set the the MIME Content-Location header of the payload		Required
FileURI	URI of the file to be loaded as a payload		Required
PayloadRef	Unique ID of the in memory XML document to be loaded as the payload		Required
MimeHeader	Set any type of MIME header name		Optional
MimeHeaderValue	Set corresponding MIME header value		Optional
SetParameter	Container for user-defined parameter to be made available to other Test Steps (source can be this message's content retrieved via XPath statement or simply a user-supplied string value)		Optional
parameterType	Choice of xpath or string		Required
Name	Name of new parameter		Required
Value	String value or XPath expression pointing to desired element/attribute value in payload, or simply a user-supplied string value		Required

7.1.6 The Dsign Operation

The DSign Operation instructs the Test Driver to digitally sign the portion of the message defined in its Reference element content.

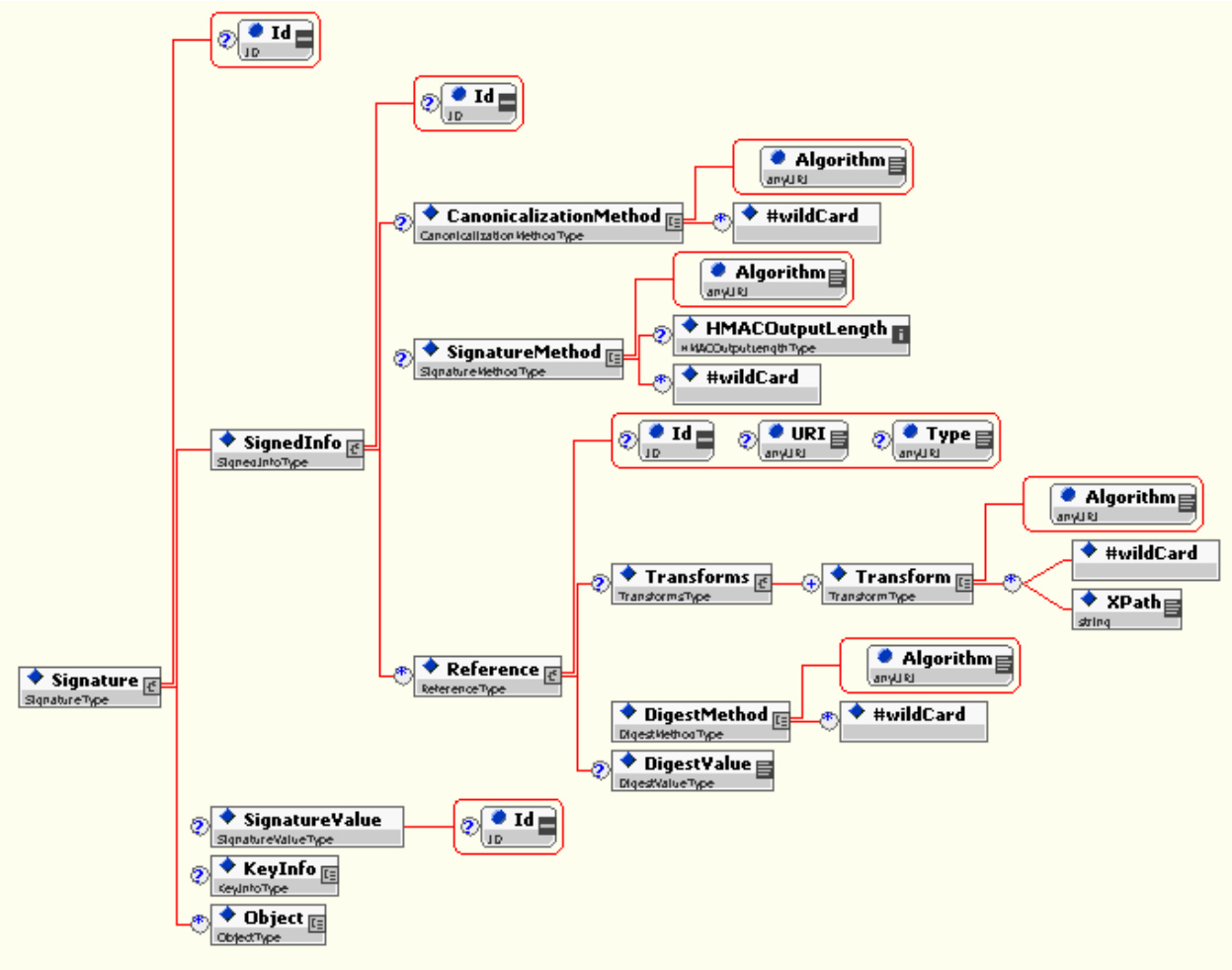


Figure 33 – Graphic representation of expanded view of the DSign element

Definition of Content

Name	Description	Default Value From Test Driver	Required/Optional
DSign	Container for Signature declaration content		Optional
ds:Signature	Signature root element, as defined in [XMLDSIG]		Required
Id	Unique identifier for Signature		Optional
SignedInfo	Create container for Canonicalizatoin and Signature algorithms and		Required

	References		
CanonicalizationMethod	Modify default container element	Container auto-generated by Test Driver	Optional
Algorithm	Modify default attribute and value	http://www.w3.org/TR/2001/REC-xml-c14n-20010315	Required
#wildCard	Generate content “inline”		Optional
SignatureMethod	Create container element		Required
Algorithm	Create attribute and value		Required
HMACOutputLength	Generate Element and its value		Optional
#wildcard	Generate content “inline”		Optional
ds:Reference	Generate container element and all default content		Optional
Id	Generate attribute and its value		Optional
URI	Modify default attribute value	“”	Optional
type	Generate attribute and its value		Optional
Transforms	Generate container relement		Optional
Transform	Generate element with its value		Optional
Algorithm	Modify default attribute value	http://www.w3.org/TR/2001/REC-xml-c14n-20010315	Required
#wildCard	Generate content “inline”		Optional
XPath	Generate element with its value		Optional
DigestMethod	Generate element with its value		Required
Algorithm	Generate attribute and value		Required
#wildCard	Generate content “inline”		Optional
DigestValue	Generate element	Set by Test Driver, based upon URI	

	with its value	value	Optional
#wildCard	Generate content “inline”		Optional
SignatureValue	Generate element and its value	Set by Test Driver at run time	Optional
Id	Generate attribute and its value		Optional
KeyInfo	Generate container Element	All required and optional content, as described in [XMLDSIG] MUST be explicitly declared (no auto-generation by Test Driver)	Optional
Object	Generate container element		Optional

7.1.7 The GetMessage Operation

The GetMessage Operation, using its child XPath Filter query retrieves a node-list of Messages received from from the Message Store of the Test Driver. The content of the node-list is dependent upon the Filter provided. The resulting node-list MAY be queried for Precondition or Test Condition tests. Any Payload associated with a message MAY be queried through the GetPayload operation.

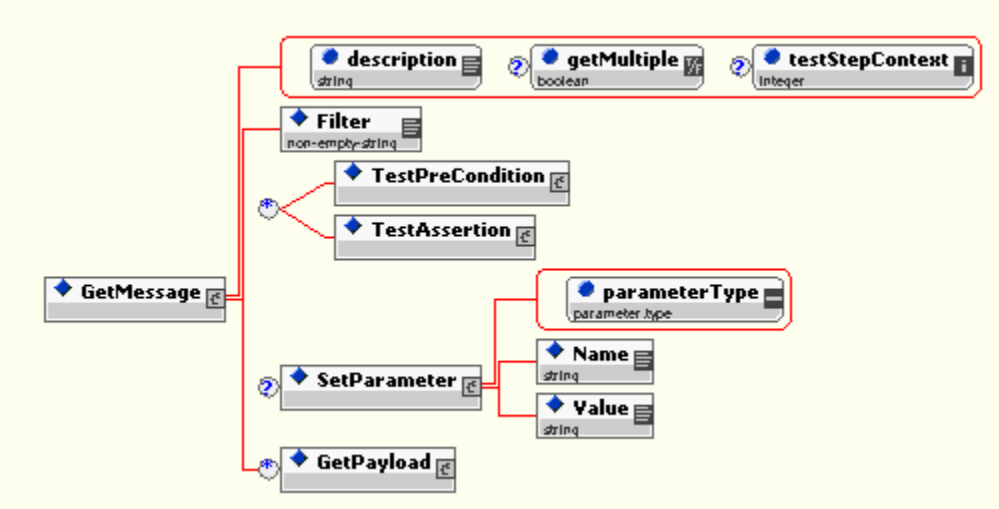


Figure 34 – Graphic representation of expanded view of the GetMessage element

Definition of Content

Name	Description	Default Value from Test Driver	Required/Optional
GetMessage	Container element for filtering, verifying and validating message and payload content		Optional
description	Description the nature of the GetMessage operation		Required
getMultiple	By default, getMultiple is “false”, indicating that only one message should be present in the node-list , even if multiple messages are in the Message Store	false	Optional
testStepContext	Integer value corresponding to previous Test Step number whose ConversationId, CPAId, MessageId and RefToMessageId is re-used		Optional
Filter	XPath query to select message(s) from Message Store		Required
TestPreCondition	Container for verification or validation operation to be performed on message as pre-condition to testing the Assertion		Optional
TestAssertion	Container for verification or validation operation to be performed to test a conformance or interoperability assertion		Optional
SetParameter	Container for user-defined parameter to be made available to other Test Steps (source can be this message’s XML content retrieved via XPath statement or simply a user-supplied string value)		Optional
parameterType	Choice of xpath or string		Required
Name	Name of new parameter		Required
Value	String value or XPath expression pointing to desired element/attribute value in message, or simply a user-supplied string value		Required

Semantics of the GetMessage operation

The Message Store is an XML document object that contains an XML representation of all synchronous and asynchronously received ebXML messages for a Test Case. The received messages for a particular Test Case MUST exist in the Message Store only for the life of the Test Case. Messages in the Message Store contain all MIME, SOAP and ebXML content, represented as an XML document. The XML format of the Message Store document MUST validate against the ebXMLMessageStore.xsd schema in appendix D

The GetMessage operation queries the Message Store document object, and retrieves (by default) the earliest (determined by Timestamp) Message element and its XML content that satisfies the XPath expression in its Filter child element. If multiple Messages satisfy the XPath Filter expression, they MAY be included in the resulting node-set by setting the “getMultiple” attribute of the GetMessage element to “true”. This is useful when a test writer wishes to count the number of “duplicate” messages received by the Test Driver for example.

A TestPreCondition or TestAssertion operation MAY query the resulting node-list generated by a GetMessage operation

In addition, new parameters that can be used by other Test Steps in this Test Case may be created using the SetParameter sub-operation. The new parameter can be used in other XPath expressions as an XPath parameter value. The SetParameter operation can select a discreet element or attribute value using the XPath expression supplied in the “Value” sub-element. Or the SetParameter operation can define a new parameter through a simple user-supplied string value assignment to the parameter.

7.1.8 The TestPreCondition Operation

The TestPreCondition Operation examines a message or messages in a GetMessage node-list by testing the content of the node-list against the VerifyContent (content value comparison) or ValidateContent (content integrity evaluation) operation.

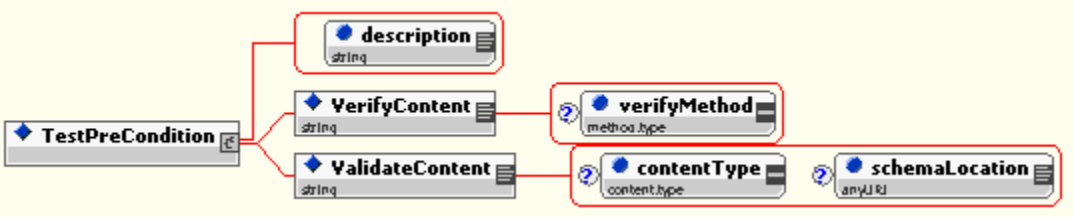


Figure 35 – Graphic representation of expanded view of the TestPreCondition element

Definition of Content

Name	Description	Default Value From Test Driver	Required/Optional
description	Metadata describing the nature of the TestPreCondition operation		Required
VerifyContent	Use XPath expression to evaluate content of message(s)		Optional

verifyMethod	Either XPath or mda-5 (for message or payload integrity test)		Optional
ValidateContent	Empty if entire XML document is to be validated or XPath expression to “point to” content to be validated for correct format if type is URI, dateTime or Signature		Optional
contentType	An enumerated list of XML, URI, time, or Signature validation descriptors		Optional
schemaLocation	URI pointing to location of schema used to validate a content type of XML		Optional

Semantics of the TestPreCondition operation

The TestPreCondition operation MUST return either a true or false result (or semantically a pass/fail result) to the Test Driver.

If TestPreCondition includes a VerifyContent sub-operation, the VerifyContent operation MUST yield a boolean value of true/false, regardless of the resulting XPath object yielded by the XPath expression. The VerifyContent XPath expression may yield a node-set, boolean, number or string object. All of these resulting objects MUST be evaluated using the “boolean” function described in [XPath]. Those evaluation rules are:

- a returned node-set object evaluates to true if and only if it is non-empty
- a returned boolean object evaluates to true if it evaluates to “true” and false if it evaluates to “false”
- a returned number object evaluates to true if and only if it is neither positive or negative zero nor NaN
- a returned string object evaluates to true if and only if its length is non-zero

If TestPreCondition includes a ValidateContent sub-operation, the ValidateContent operation MUST yield a boolean value of true/false. Rules for determining the resulting Boolean value are:

- if the contentType attribute value is XML, as defined in [XML] , the operation evaluates to true if the content at the specified XPath validates according to the schema defined in the “schemaLocation” attribute

- if the contentType is URI, as defined in [XMLSCHEMA], the operation evaluates to true if the content at the specified XPath is a valid URI
- if the contentType is dateTime, as defined in [XMLSCHEMA], the operation evaluates to true if the content af the specified XPath is a valid dateTime
- if the contentType is signature, as defined in [XMLDSIG], the operation evaluates to true if the content at the specified XPath is a validly signed element.

7.1.9 The TestAssertion Operation

The TestAssertion Operation examines a message or messages in a node-list by testing the content against an XPath expression in the TestPreCondition text. If the XPath expression returns a node-list with one or more nodes, the ConformanceCondition is “true”, else it is “false”. Within a TestAssertion Operation, content of the node-list can be further examined through the VerifyContent (content evaluation) or ValidateContent (content format evaluation).

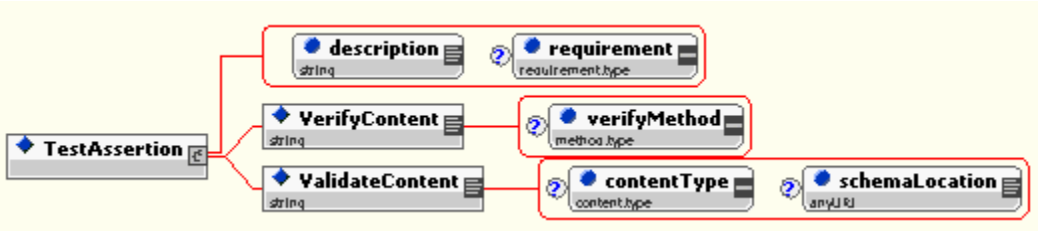


Figure 36 – Graphic representation of expanded view of the TestAssertion element

Definition of Content

Name	Description	Default Value From Test Driver	Required/Optional
description	Metadata describing the nature of the TestPreCondition operation		Required
requirement	Metadata describing whether this Conformance Condition MUST or MAY exist (for use in test reporting)		Optional
VerifyContent	Use XPath expression to evaluate content of message(s)		Optional
verifyMethod	Either XPath or mda-5 (for message or payload integrity test)		

ValidateContent	Empty if entire XML document is to be validated or XPath expression to “point to” content to be validated for correct format if type is URI, dateTime or Signature		Optionial
contentType	An enumerated list of XML, URI, dateTime, or signature validation descriptors		Optional
schemaLocation	URI describing location of validating XML schema, as defined in [XMLSCHEMA]		Optional

Semantics of the TestAssertion operation

The TestAssertion operation MUST return either a true or false result (or semantically a pass/fail result) to the Test Driver.

If TestAssertion includes a VerifyContent sub-operation, the VerifyContent operation MUST yield a boolean value of true/false, regardless of the resulting XPath object yielded by the XPath expression. The VerifyContent XPath expression may yield a node-set, boolean, number or string object. All of these resulting objects MUST be evaluated using the “boolean” function described in [XPath]. Those evaluation rules are:

- a returned node-set object evaluates to true if and only if it is non-empty
- a returned boolean object evaluates to true if it evaluates to “true” and false if it evaluates to “false”
- a returned number object evaluates to true if and only if it is neither positive or negative zero nor NaN
- a returned string object evaluates to true if and only if its length is non-zero

If TestAssertion includes a ValidateContent sub-operation, the ValidateContent operation MUST yield a boolean value of true/false. Rules for determining the resulting Boolean value are:

- if the contentType attribute value is XML, as defined in [XML], the operation evaluates to true if the content at the specified XPath validates according to the schema defined in the “schemaLocation” attribute
- if the contentType is URI, as defined in [XMLSCHEMA], the operation evaluates to true if the content at the specified XPath is a valid URI
- if the contentType is dateTime, as defined in [XMLSCHEMA], the operation evaluates to true if the content at the specified XPath is a valid dateTime
- if the contentType is signature, as defined in [XMLDSIG], the operation evaluates to true if the content at the specified XPath is a validly signed

7.1.10 The GetPayload Operation

The GetPayload operation fetches the message payload from the current message retrieved with the GetMessage operation. The message payload is retrieved based upon the required Content-ID, Content-Location or Index child element value. As with the MessageHeader, both PreCondition and TestAssertion operations can be performed on the message payload. Payload content can be verified (using the VerifyContent operation described above) using an XPath expression or by comparing its digest value against the value stored for its matching Content-Id in the current ConfigurationGroup. If the payload is an XML document, the entire document can be validated (using the ValidateContent operation described above) against the provided XML schema, or a discreet element or attribute value can be validated as a URI or dateTime.

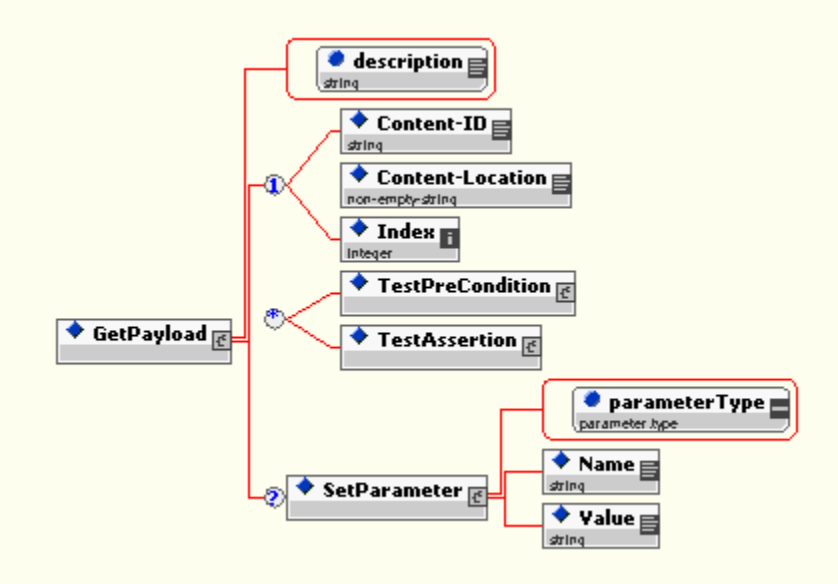


Figure 37 – Graphic representation of expanded view of the GetPayload element

Definition of Content

Name	Description	Default Value from Test Driver	Required/Optional
GetPayload	Container element for operations to verify and validate message content		Optional
description	Data describing the nature of the GetPaylod operation		Required
ContentId	Retrieve the payload using the MIME Content-ID header value	false	Optional
ContentLocation	Retrieve the payload using the MIME Content-Location value		Optional
Index	Retrieve the payload as the Nth attachment after the message envelope		Required

TestPreCondition	Container for verification or validation operation to be performed on message as pre-condition to testing the Assertion		Optional
TestAssertion	Container for verification or validation operation to be performed on message as a test of the Assertion		Optional
SetParameter	Container for user-defined parameter to be made available to other Test Steps (source can be this payload's XML content retrieved via XPath statement or simply a user-supplied string value)		Optional
parameterType	Choice of xpath or string		Required
Name	Name of new parameter		Required
Value	String value or XPath expression pointing to desired element/attribute value in payload, or simply a user-supplied string value		Required

Semantics of the GetPayload operation

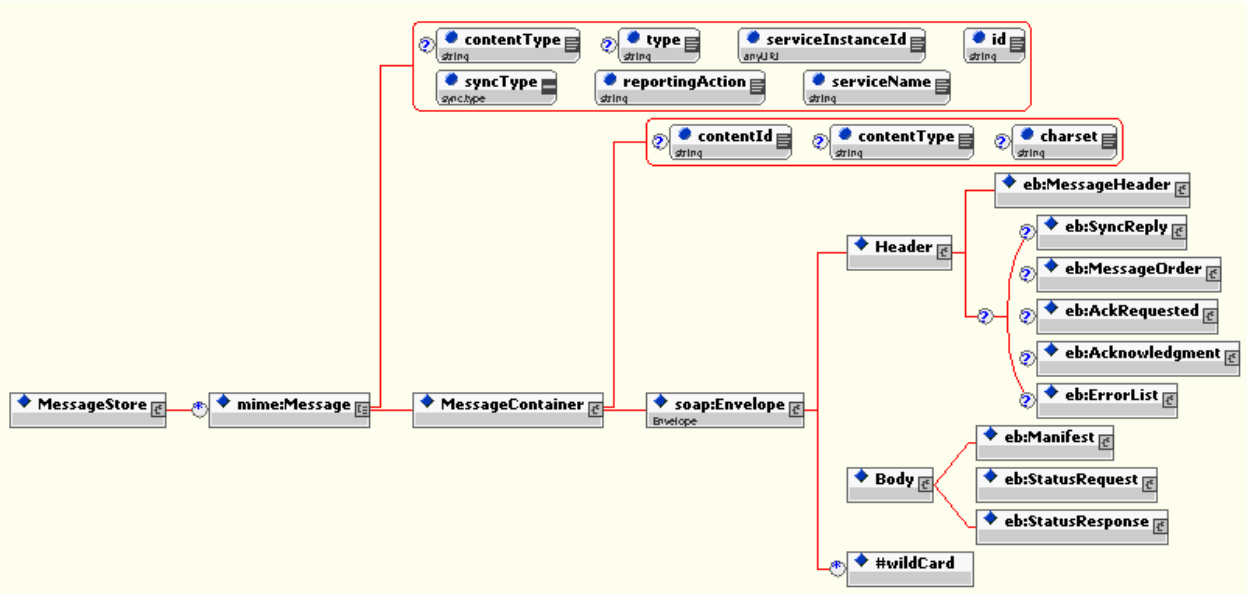
Although message payloads are not stored in the MessageStore, the Test Driver MUST be able to retrieve them for verification or validation through their corresponding Content-ID, Content-Type or Index. How the message payloads are stored by the Test Driver is implementation dependent. Unlike the GetMessage operation, which can evaluate multiple messages, the GetPayload operation can only perform a TestPreCondition or TestAssertion operation on a single payload, in a single message. That message is the "current" message retrieved by the parent GetMessage operation. If more than one message is retrieved using the GetMessage operation, the GetPayload operation will generate an exception, and the GetPayload operation will return a "fail" result. The Test Step will return a "fail" result to its parent TestCase. The failed TestCase result will be logged in the Test Report, with a failure cause of "undetermined".

All other rules regarding the TestPreCondition or TestAssertion operations previously described apply to the GetPayload operation as well.

In addition, new parameters that can be used by other Test Steps in this Test Case may be created using the SetParameter sub-operation. The new parameter can be used in other XPath expressions as an XPath parameter value. The SetParameter operation can select a discreet element or attribute value of the current payload, using the XPath expression supplied in the "Value" sub-element. Or the SetParameter operation can define a new parameter through a simple user-supplied string value assignment to the parameter.

7.1.11 Message Store Schema

The Message Store schema (Appendix D) describes the XML document format required for a Test Driver implementation. The schema facilitates a standard XPath query syntax to be used for retrieval and evaluation of received messages by the Test Driver



172-2

Figure 38 – Graphic representation of expanded view of the Test Driver MessageStore schema

Definition of Content

Name	Declaration Description	Default Value From Test Driver	Required/Optional
MessageStore	Container for XML representation of all messages received by Test Driver for a given Test Case		Required
mime:Message	Container for MIME, SOAP and ebXML message content		Optional
contentType	MIME message 'Content-Type' header		Optional
type	MIME message 'type' header		Optional
serviceInstanceId	Unique identifier for instance of Test Service that reports the message		Required
reportingAction	Action name that received the message on reporting service		Required
id	Unique identifier for this message		Required
syncType	Classifier of "synchronous" or "asynchronous"		Required
serviceName	Name of service that received the message		Required

MessageContainer	MIME SOAP messagecontainer		Required
contentId	SOAP container 'Content-ID' header		Optional
contentType	SOAP message package 'Content-Type' header		Optional
charset	SOAP message package character set		Optional
soap:Envelope	Generates container for SOAP message		Required

NOTE: All ebXML MessageStore content contained in the SOAP envelope MUST validate to the [ebMS] schema definition.

7.1.12 Service-Specific Message Payloads

The Message Payload schema (Appendix E) facilitates a standard syntax for messages sent and received by Test Service Actions that facilitates unambiguous interpretation of directives for the appropriate Action. Each possible request or response is an ebXML message payload that directs the appropriate Action defined in the ebXML MessageHeader to perform a particular function. All of the Test Framework Messages below MUST be the first payload encountered after the SOAP envelope. These Test Framework Service-Specific message payloads include:

- InitiatorRequest – XML payload content to be interpreted by the "Initiator" action to construct an ebXML Message

- PayloadVerifyResponse – XML payload content to be interpreted by the "mute" action of a Test Service. It consists of a list of Manifest "href" values, and the corresponding Boolean value indicating whether the payload(s) received by the PayloadVerify operation passed ("true") or failed ("false").

- ConfigurationRequest – XML payload content to be interpreted by the "Configurator" action of a Test Service. It consists of a ConfigurationRequest "root" element, with a configurationAction option of "query" or "replace". The default value is "replace" and indicates that configuration content for this request MUST replace the existing configuration. A configurationAction value of "query" will result in a ConfigurationResponse that only echoes back the current configuration properties of the Test Service.

- ConfigurationResponse – XML payload content to be interpred by the "mute" action of a Test Service. It consists of a ConfigurationResponse "root" element. Current CPAId, Test Service Mode, Response URL and payload digest values are returned in the XML payload.

- ErrorAppNotifyResponse – XML payload describing (using ebXML Errorlist format as defined in [EBMS]) application-level errors generated by the ErrorAppNotify action of the Test Service.

- ErrorURLNotifyResponse – XML payload describing (using ebXML Errorlist format as defined in [EBMS]) application errors generated by the ErrorURLNotify action of the Test Service.

ReportNotificationRequest – XML payload describing (using the ebXML MessageStore format defined in section 7.1.11 above) a received message. Such a message payload is followed by any additional message payloads that are part of the originally received message. Because MIME message information is not available to a Test Driver in Service Mode, only Manifest reference information (with payload Content-ID or Content-Location) will be present in the notification message.

NOTE: The ReportNotificationRequest contains three additional values besides each message item, that are obtained by Test Driver when coupled with a Test Service in reporting mode: (1) the name of the action which passed the message to test driver, (2) the test service “instanceId”, which will identify each

instance involved in a test suite and (3) the test service name that received the message.

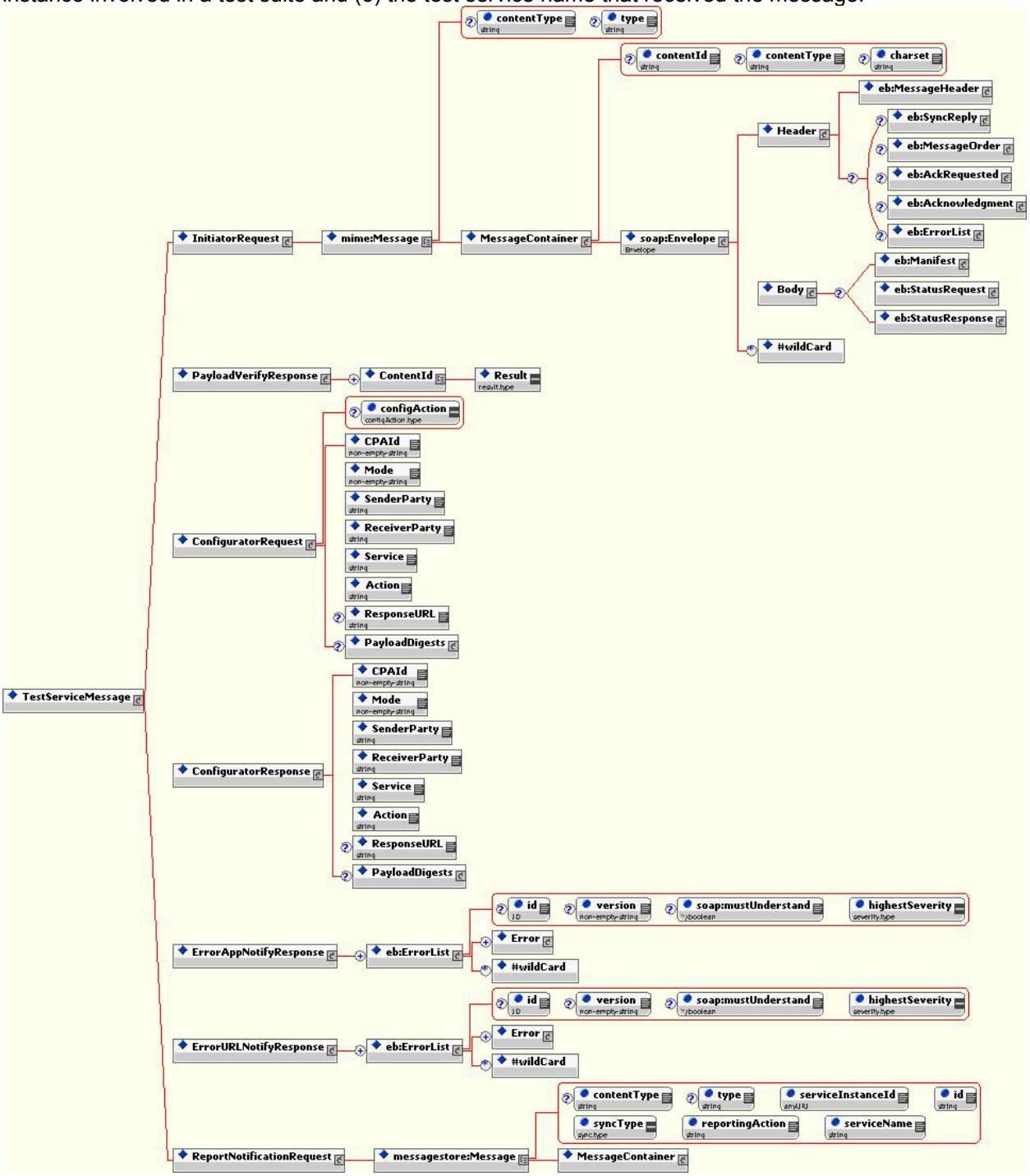


Figure 39 – Graphic representation of expanded view of the Test Driver MessagePayload.xsd schema

Definition of Content

Name	Description	Default Value	Required/Optional

TestServiceMessage	Container element for all possible test messages		Required
instanceId	Unique identifier for Test Service instance		Required
InitiatorRequest	Root element containing minimal content required for a candidate MSH to initiate a new message conversation with the Test Driver		Optional
eb:MessageHeader	ebXML Message Header declaration, as defined defined in section 7.1.4		Required
eb:SyncReply, eb:MessageOrder, eb:AckRequested, eb:Acknowledgment, eb:Manifest, eb:StatusRequest, eb:StatusResponse	SOAP header extension element declaration, as defined in section 7.1.4		Optional
PayloadVerifyResponse	Root element containing the boolean result of a payload verification test		Optional
contentId	MIME Content-ID of message verified message payload		Required
Result	Boolean result of payload verification operation		Required
ConfiguratorRequest	Root element containing minimal content required for a candidate MSH to reconfigure itself		Optional
configAction	Modify attribute to toggle between “replace” and “query” action	replace	Optional
CPAId	Reference to CPA to be used by candidate MSH		Optional
Mode	Mode of behavior for candidate MSH		Optional
ResponseURL	URL for the Test Service to send any response messages to.		Optional
NotificationURL	URL for the Test Service to send any notification messages to.		Optional
PayloadDigests	Container for Payload descriptors and verification digests		Optional
Payload	Container for individual payload descriptor content		Required

Href	Identifier for this payload		Required
Digest	Precomputed digest value for this payload		Required
ConfiguratorResponse	Root element containing minimal content required for a candidate MSH to reconfigure itself		Optional
CPAId	Reference to CPA to be used by candidate MSH		Required
Mode	Mode of behavior for candidate MSH		Required
ResponseURL	URL for the Test Service to send any response messages to		Optional
NotificationURL	URL for the Test Service to send any notification messages to		Optional
PayloadDigests	Container for Payload descriptors and verification digests		Optional
Payload	Container for individual payload descriptor content		Required
Href	Identifier for this payload		Required
Result	Boolean value for verification result for this payload: “true” (pass) or “false” (fail)		Required
ErrorAppNotifyResponse	Container for result of ErrorAppNotify action		Optional
eb:ErrorList	Errorlist content corresponding to [EBMS] specification syntax and semantics for an ebXML error		Required
ErrorURLNotifyResponse	Container for result of ErrorAppNotify action		Optional
eb:ErrorList	Errorlist content corresponding to [EBMS] specification syntax and semantics for an ebXML error		Required
ReportNotificationREquest	Container for a persistent message to be placed in the Test Driver Message Store		Optional

7.1.13 Test Report Schema

The Test Report schema (Appendix F) describes the XML report document format required for Test Driver implementations. The schema facilitates a standard XML syntax for reporting results of Test Cases and their Test Steps.

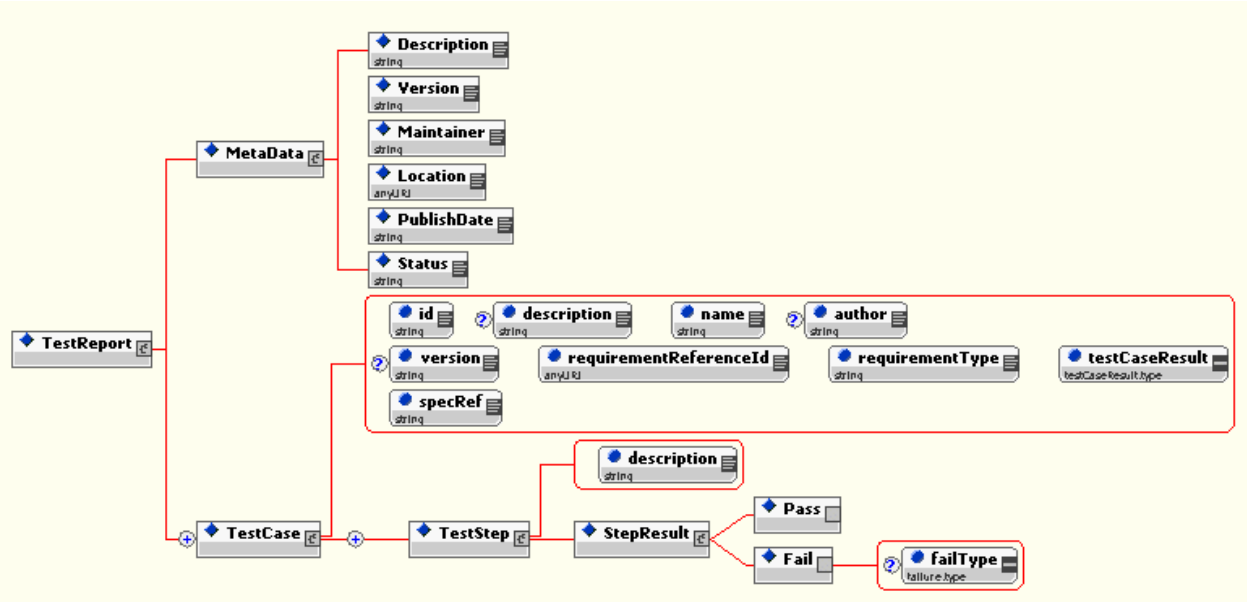


Figure 40 – Graphic representation of expanded view of the Test Driver TestReport schema

Definition of Content

Name	Description	Default Value From Test Driver	Required/Optional
TestReport	Container for all Test Case results		Required
MetaData	Container for Test Suite metadata, including Description, Version, Maintainer, Location, Publish Date and Status		Required
TestCase	Container for all result data of a single Test Case		Required
id	Unique identifier for this Test Case		Required
description	Short description of TestCase		Optional
name	Short name for Test Case		Required

author	Name of person(s) creating the Test Case		Optional
version	Version number of Test Case		Optional
requirementReferenceId	Pointer to the unique ID of the Semantic Test Requirement (in Appendix E)		Required
requirementType	Type of requirement for this Test Case (REQUIRED, OPTIONAL, RECOMMENDED, STRONGLY RECOMMENDED)		Required
testCaseResult	Enumerated Pass/Fail result for entire Test Case (pass, fail or untested)		Required
specRef	Identifier to location in specification from which this Test Case is derived		Required
TestStep	Discreet step in Test Case that MUST be evaluated in a pass/fail manner		Required
description	Short description of the function of this Test Step		Required
StepResult	Container of pass/fail result data for this Test Step		Required
Pass	Indicator that this Test Step passed		Required
Fail	Indicator that this Test Step failed		Required
failType	Enumerated type indicating type of failure (preConditionTest, assertionTest or undetermined)		Required

8 Test Material

Test material necessary to support the ebXML Testing Framework includes:

- A Testing Profile XML document
- A Test Requirements XML document
- A Test Suite XML document
- A “Basic CPA” from which variants are derived for particular tests

8.1.1 Testing Profile Document

Both conformance and interoperability testing require the creation of a Testing Profile XML document, which lists the Test Requirements against which Test Cases will be executed. A Test Profile document MUST be included in an interoperability of conformance test suite. The Testing Profile document MUST validate against the ebXMLTestProfile.xsd schema in Appendix A.

8.1.2 Test Requirements Document

Both conformance and interoperability testing require the existence of a Test Requirements document. While Test Requirements for conformance testing are specific and detailed against an ebXML specification, interoperability Test Requirements may be more generic, and less rigorous in their description and in their reference to a particular portion of an ebXML specification. However, both types of testing MUST provide a Test Requirements XML document that validates against the ebXMLTestRequirements.xsd schema in Appendix B.

8.1.3 Test Suite Document

Both conformance and interoperability testing require the existence of a Test Suite XML document that validates against the ebXMLTestSuite.xsd schema in Appendix C. It is important to note that test case scripting inside the Test Suite document MUST take into account the test harness architecture. Although MIME and SOAP message content can be manipulated by a Test Driver in Connection Mode, such content may not be accessible by a Test Driver in Service Mode, as the MSH does not communicate this data to the application layer. Therefore, the following test scripting rules SHOULD be followed when designing Test Cases:

- When the message material is to be sent or analyzed at by a Test Driver in Service Mode (i.e. the Test Driver acts as an application component), MIME header and SOAP content SHOULD NOT be declared (in a PutMessage operation) or queried (in a GetMessage operation). However, for the sake of uniform scripting, a Test Driver that conforms to this specification MUST accept MIME message envelope and SOAP header material defined in the declaration of the PutMessage

operation (it will then ignore superfluous elements when passing the message to the Test Service). “Accepting” means: (1) when sending (e.g. via PutMessage), MIME and SOAP envelope material will be ignored when invoking the Initiator service action, (2) when receiving, any filtering condition or reference to MIME and SOAP material will be ignored, e.g. removed from the set of conditions used in a GetMessage step. In addition, a Test Driver that conforms to this specification MUST accept XPath query expressions that reference MIME and SOAP message content, even though such content MAY not be included in the MessageStore representation of a message.

- When the message material is to be sent or analyzed in Connection Mode (i.e. the Test Driver acts as an MSH component), MIME header and SOAP content MAY be declared (in a PutMessage operation) or queried (in a GetMessage operation). At this messaging level, all message data is acessable by the Test Driver.

In th graphical example of the above scenarios, an [ebMS] interoperability test suite will generally require messages to be generated and received at application level (Service Mode) directly from Test Driver to Test Service as illustrated in Figure 5. In contrast, an ebMS conformance test suite which will require messages to be generated and received at transport level (Connection Mode), as illustrated in Figures 2 and 3.

8.1.4 Base CPA and derived CPAs

Both conformance and interoperabililty testing require the existence of a “base CPA” configuration that describes both the Test Driver and Test Service Collaboration Protocol Profile Agreement. This is the “bootstrap” configuration for all messaging between the testing and candidate ebXML applications. How the base CPA is represented to the applications is implementation specific, however the base CPA configuration MUST be semantically equivalent to the CPA defined in Conformance or Interoperability Test Suite Specification.

Modified (or derived) versions of the base CPA MUST have unique CPAIds identifying them as derivations of the base CPA to both the Test Driver and Test Service. A Test Harness implementation MAY reference CPAs that are derived from the base CPA in order to perform a particular type of conformance or interoperability test. CPA’s derived from the base CPA and used in a Test Suite MUST be documented in the appropriate ebXML Conformance Test Suite or ebXML Interoperability Test Suite Specification. The unique CPAId, and the list of discreet variations from the base CPA MUST are included in the Conformance or Interoperability Test Suite document.

9 Test Material Examples

This section includes example test material to illustrate

- A Test Requirements Document – Listing all Test Requirements for an ebXML implementation
- A Test Profile Document – Listing all selected Test Requirements to be exercised
- A Test Suite Document – Listing all Executable Test Cases for an ebXML implementation

9.1 Example Test Requirements

Below are two XML documents illustrating how Test Requirements are constructed, in this case for an ebXML MS 2.0 implementation. In this particular case, the two documents represent Conformance and Interoperability Test Requirements for an ebXML Messaging Services V2.0 implementation. The example XML documents below include a subset of testing requirements defined for implementations of the ebXML Messaging Services v2.0 Specification. Each Test Requirement may have one or more Functional Requirements that together must be satisfied in order for an implementation to fully meet that Test Requirement.

9.1.1 Conformance Test Requirements

In the example below, a “packaging” TestRequirement element contains two FunctionalRequirement elements. The first Functional Requirement states that the primary SOAP message MUST be the first MIME part of the message. The second packaging Functional Requirement states that the Content-Type MIME header of the Message Package MUST be “text/xml”. If all Test Cases having a requirement reference to these two Functional Requirements “pass”, then an ebXML MS v2.0 implementation would be deemed “conformant” to the specification for the “Packaging” of ebXML messages. Of course, this is a limited set of Test Requirements for illustrative purposes only.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Requirements xmlns="http://www.oasis-open.org/tc/ebxml-iic/conformance/reqs"
xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
xsi:schemaLocation="http://www.oasis-open.org/tc/ebxml-iic/conformance/reqs/
ebXMLTestRequirements.xsd">
<MetaData>
  <Description>Master Requirements File: ebXML Messaging Services 2.0</Description>
  <Version>1.0</Version>
  <Maintainer>Michael Kass<Michael.kass@nist.gov></Maintainer>
  <Location>http://www.oasis-open.org/commitees/ebxml-
iic/ebmsg/requirements1.0.xml</Location>
  <PublishDate>20 Feb 2003</PublishDate>
  <Status>DRAFT</Status>
  </MetaData>
<!—Main Test Requirement, for message packaging→
<TestRequirement id="req_id_2" name="PackagingSpecification" specRef="ebMS-2#2.1"
functionalType="packaging">
<!—Define first sub-requirement to fulfill packaging testing→
```

```
<FunctionalRequirement id="funreq id 2"
name="GenerateConformantSOAPWithAttachMIMEHeaders" specRef="ebMS-2#2.1.2">
<Clause>
<!–Set first condition of the message is of type “multipart-mime” →
  <Condition id="condition_id_2" requirementType="required">For each generated mesage,
if it is multipart MIME</Condition>
  <Or />
<!–Set alternate condition that the message is not “text/xml” →
  <Condition id="condition_id_305" requirementType="required">if it is not
text/xml</Condition>
  </Clause>
<!–Define the Assertion that the first part of message is a SOAP message →
  <Assertion id="assert_id_2" requirementType="required">The primary SOAP message is
carried in the root body part of the message.</Assertion>
  </FunctionalRequirement>
<!–Define a second sub-requirement to fulfill packaging testing→
<FunctionalRequirement id="funreq id 4" name="GenerateCorrectMessagePackageContent-Type"
specRef="ebMS-2#2.1.2">
<Clause>
<!–Define condition that the candidate MSH generates a message →
  <Condition id="condition_id_4" requirementType="required">For each generated
message</Condition>
  </Clause>
<!–Define the Assertion that the Content-Type of MIME header of that message is
“text/xml” →
  <Assertion id="assert_id_4" requirementType="required">The Content-Type MIME header in
the Message Package contains a type attribute of "text/xml".</Assertion>
  </FunctionalRequirement>
</TestRequirement>
<!–Define a new Test Requirement, for  the Core Extension Elements of messaging→
<TestRequirement id="req_id_3" name="CoreExtensionElements" specRef="ebMS-2#3.1.1"
functionalType="packaging">
<!–Define a sub-requirement to test the CPAId extension element→
<FunctionalRequirement id="funreq id 35" name="ReportFailedCPAIDResolution"
specRef="ebMS-2#3.1.2">
<Clause>
<!–First , set condition of a candidate MSH  receiving a message with an unresolvable
CPAId→
  <Condition id="condition_id_40" requirementType="required">For each received message,
if value of the CPAId element on an inbound message cannot be resolved</Condition>
  </Clause>
<!–Next , define the Assertion that the candidate MSH MUST ( since requirementType is
“required”)  respond with an Error→
  <Assertion id="assert_id_35" requirementType="required">The MSH responds with an error
(ValueNotRecognized/Error).</Assertion>
  </FunctionalRequirement>
<!–Define a sub-requirement to test continuity in message ConversationId→
<FunctionalRequirement id="funreq_id_36" name="ProvideConversationIdIntegrity"
specRef="ebMS-2#3.1.3">
<Clause>
<!–First , set condition of all messages generated by a Candidate Implementation
pertaining to a single CPAId→
  <Condition id="condition_id_41" requirementType="required">For each generated message
within the context of the specified CPAId</Condition>
  </Clause>
<!–Next , define the Assertion that a ConversationId element is always present→
  <Assertion id="assert_id_36" requirementType="required">The generated ConversationId
will be present in all messages pertaining to the given conversation.</Assertion>
  </FunctionalRequirement>

</TestRequirement>
</Requirements>
```

9.1.2 Interoperability Test Requirements

In the example below, a “basic interoperability profile” TestRequirement element contains two FunctionalRequirement elements. The first Functional Requirement states that ebXML MS implementation MUST be able to receive and send a basic ebXML message without a payload. The second packaging Functional Requirement states that an ebXML MS implementation MUST be able to process and return a simple ebXML message with one payload. If all Test Cases having a requirement reference to these two Functional Requirements “pass”, then an ebXML MS v2.0 implementation would be deemed “interoperable” to the Basic Interoperability Profile Specification for ebXML Messaging. Of course, this is a limited set of Test Requirements for illustrative purposes only.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Requirements xmlns="http://www.oasis-open.org/tc/ebxml-iic/interop/reqs"
xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
xsi:schemaLocation="http://www.oasis-open.org/tc/ebxml-iic/interop/reqs
ebXMLTestRequirements.xsd">
<MetaData>
  <Description>Interoperability Requirements File: ebXML Messaging Services
2.0</Description>
  <Version>1.0</Version>
  <Maintainer>Michael Kass <michael.kass@nist.gov></Maintainer>
  <Location>http://www.oasis-open.org/commitees/ebxml-
iic/ebmsg/ms_2.0_interop_requirements1.0.xml</Location>
  <PublishDate>11 Feb 2003</PublishDate>
  <Status>DRAFT</Status>
  </MetaData>
<!—Main Test Requirement, for basic interoperability testing→
<TestRequirement id="req id 1" name="Basic Interoperability Profile" specRef="MS 2.0 BIP
0.8" functionalType="basic interoperability">
<!—Define first sub-requirement to fulfill basic testing, sending a “no payload”
message→
<FunctionalRequirement id="funreq_id_1" name="BasicExchangeNoPayload" specRef="ebMS 2.0
BIP#3.2.1">
<Clause>
<!—First , set condition of a candidate MSH  receiving a message with no payload→
  <Condition id="condition_id_1" requirementType="required">For each received ebXML
message with no payload, received by the “Dummy” action</Condition>
  </Clause>
<!—Next , define the Assertion of expected behavior for the Dummy Action→
  <Assertion id="assert_id_1" requirementType="required">The message is received and
processed, and a simple response message is returned</Assertion>
  </FunctionalRequirement>
<!—Define second sub-requirement to fulfill basic testing, sending a “one payload”
message→

<FunctionalRequirement id="funreq id 2" name="BasicExchangeOnePayload" specRef="ebMS 2.0
BIP#3.2.2">
<Clause>
<!—Set condition of a candidate MSH  receiving a message with one payload→
  <Condition id="condition_id_2" requirementType="required">For each received ebXML
message with one payload, received by the “Reflector” action </Condition>
  </Clause>
<!—Define the Assertion of expected behavior for the Reflector Action→
  <Assertion id="assert_id_2" requirementType="required">The message is received and
processed, and a simple response message with the identical  payload is
returned</Assertion>
  </FunctionalRequirement>
<!—Define third sub-requirement to fulfill basic testing, sending a “three payload”
message→
<FunctionalRequirement id="funreq_id_3" name="BasicExchangeThreePayloads" specRef="ebMS
2.0 BIP#3.2.3">
<Clause>
<!—Set condition of a candidate MSH  receiving a message with three payloads→
  <Condition id="condition_id_3" requirementType="required">For each received ebXML
message with three payloads, received by the “Reflector” action</Condition>
```

```
  </Clause>
<!—Define the Assertion of expected behavior for the Reflector Action→
  <Assertion id="assert_id_3" requirementType="required">The message is received and
processed, and a simple response message with the identical three payloads are
returned</Assertion>
  </FunctionalRequirement>
<!—Define third sub-requirement to fulfill basic testing, generating Error messages→
<FunctionalRequirement id="funreq id 4" name="BasicExchangeGenerateError" specRef="ebMS
2.0 BIP#3.2.4">
<Clause>
<!—Set condition of a candidate MSH  receiving an erroneous message→
  <Condition id="condition_id_4" requirementType="required">For each received basic
ebXML message that should generate  an Error </Condition>
  </Clause>
<!—Define the Assertion of expected behavior for the candidate MSH  →
  <Assertion id="assert_id_4" requirementType="required">The message is received and,
the MSH returns a message to the originating party with an ErrorList and appropriate
Error message </Assertion>
  </FunctionalRequirement>
</TestRequirement>
</Requirements>
```

9.2 Example Test Profiles

Below are two XML documents illustrating how a Test Profile document is constructed, in this case for an ebXML MS v2.0 implementation. The example XML documents below represent a subset of test requirements to be exercised. The Test Profile document provides a list of ID references (pointers) to Test Requirements or Functional Requirements in an external Test Requirements document (see above). A Test Harness would read this document, resolve the location of the Test Requirements document, and then execute all Test Cases in the Test Suite document that point to (via ID reference) the Test Requirements listed below. Note that a Test Driver can execute Test Cases pointing to a Functional Requirement (discreet requirement) or a Test Requirement (a container of a group of Functional Requirements). If the TestRequirementRef id attribute value points to a Test Requirement, then all Test Cases for all child Functional Requirements will be executed by the Test Harness (This is a way to conveniently execute a cluster of Test Cases by specifying a single Test Requirement.). This method is used for both conformance and interoperability testing.

9.2.1 Conformance Test Profile Example

The Test Profile document below would be used to drive a Test Harness, by executing all Test Cases that point (via ID) to the listed Test Requirement references (including individual Functional Requirements and a single Test Requirement listed in the above example Conformance Test Requirements document.

```
<?xml version="1.0" encoding="UTF-8" ?>
<TestProfile xmlns="http://www.oasis-open.org/tc/ebxml-iic/test-profile"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.oasis-
open.org/tc/ebxml-iic/test-profile http://www.oasis-open.org/tc/ebxml-iic/test-
profile/test-profile.xsd" requirementsLocation="ebxml-iic-msg-v20-conformance_reqs.xml"
name="ebXML MS v2.0 Conformance Test Requirements" description="Core conformance testing
profile for ebXML MS v2.0 implementations”>
  <TestRequirementRef id="funreq_id_2" />  <!—Execute all Test Casses that reference the
Basic SOAP message structure Functional Requirement→
```

```
  <TestRequirementRef id="funreq_id_4" /> <!—Execute all Test Cases that reference Message
Packaeg Content Type Functional Requirement→
  <TestRequirementRef id="req_id_2" />  <!—Execut all Test Cases that reference all
Functional Requirements within the  Core Extension Elements Test Requirement→
  </TestProfile>
```

9.2.2 Interoperability Test Profile Example

The Test Profile document below would be used to drive a Test Harness, by executing all Test Cases that point (via ID) to the listed Test Requirement references (including individual Functional Requirements and a single Test Requirement listed in the above example Interoperability Test Requirements document.

```
<?xml version="1.0" encoding="UTF-8" ?>
<TestProfile xmlns="http://www.oasis-open.org/tc/ebxml-iic/test-profile"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.oasis-
open.org/tc/ebxml-iic/test-profile http://www.oasis-open.org/tc/ebxml-iic/test-
profile/test-profile.xsd" requirementsLocation="ebxml-iic-msg-v20-conformance_reqs.xml"
name="ebXML MS v2.0 Conformance Test Requirements" description="Core conformance testing
profile for ebXML MS v2.0 implementations”>
  <TestRequirementRef id="funreq_id_1.1" /> <!—Execute all Test Casses that reference the
“Basic Exchange, No Payload” Functional Requirement→
  <TestRequirementRef id="funreq_id_1.2" /> <!—Execute all Test Casses that reference the
“Basic Exchange, One Payload” Functional Requirement→
  </TestProfile>
```

9.3 Example Test Suites

Below are two XML documents illustrating how Test Cases are constructed, in this case for testing an ebXML MS v2.0 implementation. Each Test Case has a required “requirementReferenceId” attribute, pointing to a Functional Requirement in the Test Requirements document. A Test Driver executes all Test Cases in this document that have a requirementReferenceId value matching the particular Semantic Test Requirement being exercised.

9.3.1 Conformance Test Suite

In the example below, a series of four Test Cases make up a Test Suite. A Test Driver executing conformance Test Cases operates in “connection” mode, meaning it is not interfaced to any MSH, and is acting on its own. Each Test Case exercises a Functional Requirement listed in section 10.1 The Test Cases below do the following:

- Send a message and elicit a response message that is verified as a SOAP message

- Verifies that an elicited response message content type is “text/xml”
- Verifies that an ebXML Error is returned in a response message when an unresolvable CPAId is received
- Verifies that the ConversationId element is present in a simple response message

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
   EbXML Messaging v2 Conformance  Test SuiteSample  Instance File.
   Michael Kass <michael.kass@nist.gov>.
   Date:  02/20/03
--

<!—Define Test Suite, with configuration reference to  “bootstrap” Test Driver→
 <ebTest:TestSuite ebTest:configurationGroupRef="cpa_basic" xmlns:ebTest="http://www.oasis-
open.org/tc/ebxml-iic/tests" xmlns:xpath="http://www.oasis-open.org/tc/ebxml-iic/xpath"
xmlns:mime="http://www.oasis-open.org/tc/ebxml-iic/tests/mime"
xmlns:soap="http://www.oasis-open.org/tc/ebxml-iic/tests/soap"
xmlns:eb="http://www.oasis-open.org/tc/ebxml-iic/tests/eb" xmlns:tns="http://www.oasis-
open.org/tc/ebxml-iic/tests/tns" xmlns:xlink="http://www.oasis-open.org/tc/ebxml-
iic/tests/xlink" xmlns:cfg="http://www.oasis-open.org/tc/ebxml-iic/tests/config"
xmlns:ds="http://www.oasis-open.org/tc/ebxml-iic/tests/xmldsig"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.oasis-
open.org/tc/ebxml-iic/tests ebXMLTestSuite.xsd">
<!—Document Test Suite for later use in Test Reporting→
 <ebTest:MetaData>
  <ebTest:Description>Conformance Test Suite File: ebXML Messaging Services
2.0</ebTest:Description>
  <ebTest:Version>1.0</ebTest:Version>
  <ebTest:Maintainer>Michael Kass <michael.kass@nist.gov></ebTest:Maintainer>
  <ebTest:Location>http://www.oasis-open.org/commitees/ebxml-
iic/ebmsg/conf_testsuite1.0.xml</ebTest:Location>
  <ebTest:PublishDate>12 February 2003</ebTest:PublishDate>
  <ebTest:Status>DRAFT</ebTest:Status>
  </ebTest:MetaData>
<!—Define basic “bootstrap” configuration data for this Test Suite →
 <ebTest:ConfigurationGroup ebTest:id="cpa_basic">
  <ebTest:CPAId>cpa_basic</ebTest:CPAId>
  <ebTest:Mode>connection</ebTest:Mode>
  <ebTest:SenderParty>urn:oasis:iic:testdriver</ebTest:SenderParty>
  <ebTest:ReceiverParty>urn:oasis:iic:testservice</ebTest:ReceiverParty>
  <ebTest:Service>urn:ebXML:iic:test.</ebTest:Service>
  <ebTest:Action>Dummy</ebTest:Action>
  </ebTest:ConfigurationGroup>
<!—Define Test Case,referencing corresponding Functional Requirement in Conformance Test
Requirements XML document →
 <ebTest:TestCase ebTest:requirementReferenceId="urn:Funreq:2" ebTest:id="urn:TestCase:id:2"
ebTest:description="SOAP message must be in root part of MIME message">
 <ebTest:TestStep ebTest:mode="connection">
 <ebTest:PutMessage ebTest:description="Send basic message header">
 <ebTest:MessageDeclaration> <!—Declare a basic ebXML message →
 <mime:Message>
 <mime:MessageContainer>
 <soap:Envelope>
 <soap:Header>
 <eb:MessageHeader> <!—Declare MessageHeader, using default ConfigurationGroup parameter values
for Action and CPAId  ( Dummy Action, basic CPA )→

  </eb:MessageHeader>
  </soap:Header>
```

```
  </soap:Envelope>
  </mime:MessageContainer>
  </mime:Message>
  </ebTest:MessageDeclaration>
  </ebTest:PutMessage>
  </ebTest:TestStep>
 <ebTest:TestStep >
 <ebTest:GetMessage ebTest:description="Correlate returned message">
<!—Filter returned messages from MessageStore, using RefToMessageId to find the desired message →
  <ebTest:Filter>eb:CPAId='cpa_basic' and eb:ConversationebTest:id=$ConversationId and
eb:MessageData/eb:RefToMessageebTest:id=$RefToMessageId</ebTest:Filter>
 <ebTest:TestAssertion ebTest:description="Verify that an SOAP Message is found in the root
part of the MIME message">
<!--Use XPath to verify correct message structure--
→<ebTest:VerifyContent>/mime:Message[mime:MessageContainer[1]/soap:Envelope]</ebTest:
VerifyContent>
  </ebTest:TestAssertion>
  </ebTest:GetMessage>
  </ebTest:TestStep>
  </ebTest:TestCase>
<!--Define Test Case,referencing corresponding Functional Requirement in Conformance
Test Requirements XML document -->
 <ebTest:TestCase ebTest:requirementReferenceId="urn:funreq:4" ebTest:id="urn:TestCase:id:4"
ebTest:description="Message package Content-Type is text/xml">
 <ebTest:TestStep >
 <ebTest:PutMessage ebTest:description="Send basic message header">
 <ebTest:MessageDeclaration> <!—Declare a basic ebXML message →
 <mime:Message>
 <mime:MessageContainer
 <soap:Envelope>
 <soap:Header>
 <eb:MessageHeader><!—Declare MessageHeader, using default ConfigurationGroup parameter values
for Action and CPAId  ( Dummy Action, basic CPA )→

  </eb:MessageHeader>
  </soap:Header>
  </soap:Envelope>
  </mime:MessageContainer>
  </mime:Message>
  </ebTest:MessageDeclaration>
  </ebTest:PutMessage>
  </ebTest:TestStep>
 <ebTest:TestStep >
 <ebTest:GetMessage ebTest:description="Correlate returned message">
<!—Filter returned messages from MessageStore, using RefToMessageId to find the desired message →
  <ebTest:Filter>eb:CPAId='cpa_basic' and eb:ConversationebTest:id=$ConversationId and
eb:MessageData/eb:RefToMessageebTest:id=$RefToMessageId</ebTest:Filter>
 <ebTest:TestAssertion ebTest:description="Verify message package Content-type">
<!—Use XPath to verify that MIME Content-Type is "text/xml" →
  <ebTest:VerifyContent>/mime:Message[mime:MessageContainer[0] and (@Content-Type =
'text/xml)')]</ebTest:VerifyContent>
  </ebTest:TestAssertion>
  </ebTest:GetMessage>
  </ebTest:TestStep>
  </ebTest:TestCase>
<!—Define Test Case,referencing corresponding Functional Requirement in Conformance Test
Requirements XML document →
 <ebTest:TestCase ebTest:requirementReferenceId="Funreq:35" ebTest:id="urn:TestCase:id:35"
ebTest:description="If CPAId cannot be resolved, respond with ValueNotRecognized Error">
 <ebTest:TestStep >
 <ebTest:PutMessage ebTest:description="Declare  message  with  and CPAId set to 'null'">
 <ebTest:MessageDeclaration>
 <mime:Message>
```

```
 <mime:MessageContainer>
 <soap:Envelope>
 <soap:Header>
<!—Override default configuration value for CPAId in ebXML MessageHeader element and override the
default CPAId,  using instead an unresolvable CPAId →
 <eb:MessageHeader>

  <eb:CPAId>null</eb:CPAId>
  </eb:MessageHeader>
  </soap:Header>
  </soap:Envelope>
  </mime:MessageContainer>
  </mime:Message>
  </ebTest:MessageDeclaration>
  </ebTest:PutMessage>
  </ebTest:TestStep>
 <ebTest:TestStep >
 <ebTest:GetMessage ebTest:description="Correlate returned messages">
<!—Filter returned messages from MessageStore, using RefToMessageId to find the desired message →
  <ebTest:Filter>eb:CPAId='cpa_basic' and eb:ConversationebTest:id=$ConversationId and
eb:MessageData/RefToMessageebTest:id=$RefToMessageId</ebTest:Filter>
<!—Use XPath to verify that an Error element with the correct errorCode attribute value
is present in the message →
 <ebTest:TestAssertion ebTest:description="Verify that Error is returned">
 <ebTest:VerifyContent>/mime:Message[mime:MessageContainer[1]/soap:Envelope/soap:Hea
der/eb:ErrorList/eb:Error[@eb:errorCode = 'ValueNotRecognized' and @eb:severity =
'Error']]</ebTest:VerifyContent>
  </ebTest:TestAssertion>
  </ebTest:GetMessage>
  </ebTest:TestStep>
  </ebTest:TestCase>
<!—Define Test Case,referencing corresponding Functional Requirement in Conformance Test
Requirements XML document →
 <ebTest:TestCase ebTest:requirementReferenceId="Funreq:36" ebTest:id="urn:TestCase:id:36"
ebTest:description="ConversationId is always present">
 <ebTest:TestStep >
 <ebTest:PutMessage ebTest:description="Send basic Dummy message header">
 <ebTest:MessageDeclaration> <!—Declare a basic ebXML message →
 <mime:Message>
 <mime:MessageContainer>
 <soap:Envelope>
 <soap:Header>
 <eb:MessageHeader>

  </eb:MessageHeader>
  </soap:Header>
  </soap:Envelope>
  </mime:MessageContainer>
  </mime:Message>
  </ebTest:MessageDeclaration>
  </ebTest:PutMessage>
  </ebTest:TestStep>
 <ebTest:TestStep >
 <ebTest:GetMessage ebTest:description="Correlate returned messages">
  <ebTest:Filter>eb:CPAId='cpa_basic' and eb:ConversationebTest:id=$ConversationId and
eb:MessageData/RefToMessageebTest:id=$RefToMessageId</ebTest:Filter>
 <ebTest:TestAssertion ebTest:description="Verify that Conversation Id is not present">
 <!—Use XPath to verify the XML content, and therefore the Assertion →
<ebTest:VerifyContent>/mime:Message[mime:MessageContainer[1]/soap:Envelope/soap:Head
er/eb:MessageHeader/eb:ConversationId]</ebTest:VerifyContent>
  </ebTest:TestAssertion>
  </ebTest:GetMessage>
  </ebTest:TestStep>
```

```
  </ebTest:TestCase>
  </ebTest:TestSuite>
```

9.3.2 Interoperability Test Suite

In the example below, a series of four Test Cases make up an Interoperability Test Suite. A Test Driver executing conformance Test Cases operates in “service” mode, meaning it is interfaced to a MSH. Each Test Case exercises a Functional Requirement listed in section 10.2 The Test Cases below do the following:

- Perform a basic message exchange with no message payload
- Verify integrity of 1 payload in round-trip message transmission
- Verify integrity of 3 payloads in round-trip message transmission
- Perform a basic message exchange with a returned Error message

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
   EbXML Messaging v2 Interop Test SuiteSample  Instance File.
   Michael Kass <michael.kass@nist.gov>.
   Date: 02/20/03
  -->
<!—Define Test Suite, with configuration reference to  “bootstrap” Test Driver→
 <ebTest:TestSuite ebTest:configurationGroupRef="cpa basic"
xmlns:ebTest="http://www.oasis-open.org/tc/ebxml-iic/tests"
xmlns:xpath="http://www.oasis-open.org/tc/ebxml-iic/xpath" xmlns:mime="http://www.oasis-
open.org/tc/ebxml-iic/tests/mime" xmlns:soap="http://www.oasis-open.org/tc/ebxml-
iic/tests/soap" xmlns:eb="http://www.oasis-open.org/tc/ebxml-iic/tests/eb"
xmlns:tns="http://www.oasis-open.org/tc/ebxml-iic/tests/tns"
xmlns:xlink="http://www.oasis-open.org/tc/ebxml-iic/tests/xlink"
xmlns:cfg="http://www.oasis-open.org/tc/ebxml-iic/tests/config"
xmlns:ds="http://www.oasis-open.org/tc/ebxml-iic/tests/xmldsig"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.oasis-open.org/tc/ebxml-iic/tests/ebXMLTestSuite.xsd">
<!—Document Test Suite for later use in Test Reporting→
 <ebTest:MetaData>
  <ebTest:Description>Interoperability Test Suite File: ebXML Messaging Services
2.0</ebTest:Description>
  <ebTest:Version>1.0</ebTest:Version>
  <ebTest:Maintainer>Michael Kass <michael.kass@nist.gov></ebTest:Maintainer>
  <ebTest:Location>http://www.oasis-open.org/commitees/ebxml-
iic/ebmsg/interop_testsuite1.0.xml</ebTest:Location>
  <ebTest:PublishDate>12 February 2003</ebTest:PublishDate>
  <ebTest:Status>DRAFT</ebTest:Status>
  </ebTest:MetaData>
<!—Define basic “bootstrap” configuration data for this Test Suite →
 <ebTest:ConfigurationGroup ebTest:id="cpa_basic">
  <ebTest:CPAId>cpa_basic</ebTest:CPAId>
  <ebTest:Mode>connect</ebTest:Mode>
  <ebTest:SenderParty>urn:oasis:iic:testdriver</ebTest:SenderParty>
  <ebTest:ReceiverParty>urn:oasis:iic:testservice</ebTest:ReceiverParty>
  <ebTest:Service>urn:ebXML:iic:test.</ebTest:Service>
  <ebTest:Action>Dummy</ebTest:Action>
<!—Predefine message payload MDA-5 digest values for use in payload verification Test
Cases →
 <ebTest:PayloadDigests>
 <ebTest:Payload>
  <ebTest:Href>cid:payload_1</ebTest:Href>
  <ebTest:Digest>abc</ebTest:Digest>
  </ebTest:Payload>
```

```
 <ebTest:Payload>
  <ebTest:Href>cid:payload_2</ebTest:Href>
  <ebTest:Digest>def</ebTest:Digest>
  </ebTest:Payload>
 <ebTest:Payload>
  <ebTest:Href>cid:payload_3</ebTest:Href>
  <ebTest:Digest>ghi</ebTest:Digest>
  </ebTest:Payload>
  </ebTest:PayloadDigests>
  </ebTest:ConfigurationGroup>
<!—Define "alternate" configuration data for use in particular Test Cases in this Test
Suite →
 <ebTest:ConfigurationGroup ebTest:id="cpa_basic_no_key_info">
  <ebTest:CPAId>cpa_basic</ebTest:CPAId>
  <ebTest:Mode>connect</ebTest:Mode>
  <ebTest:SenderParty>urn:</ebTest:SenderParty>
  <ebTest:ReceiverParty>urn:</ebTest:ReceiverParty>
  <ebTest:Service>whatever</ebTest:Service>
  <ebTest:Action>whatever</ebTest:Action>
  </ebTest:ConfigurationGroup>
<!—Declare XML Message Payload content for use in particular Test Cases in this Test
Suite→
 <ebTest:Payload ebTest:id="payload_1">
  <Message name="payload_1" />
  </ebTest:Payload>
 <ebTest:Payload ebTest:id="payyload_2">
  <Message name="payload_2" />
  </ebTest:Payload>
 <ebTest:Payload ebTest:id="payload_3">
  <Messagename="payload_3" />
  </ebTest:Payload>
<!—Define Test Case,referencing corresponding Functional Requirement in Interoperability
Test Requirements XML document →
 <ebTest:TestCase ebTest:requirementReferenceId="funreq_id_1.1"
ebTest:id="urn:TestCase:id:1.1" ebTest:description="Basic exchange, no payload">
 <ebTest:TestStep >
 <ebTest:PutMessage ebTest:description="Send basic message header">
 <ebTest:MessageDeclaration>
 <mime:Message>
 <mime:MessageContainer>
 <soap:Envelope>
 <soap:Header>
<!—Declare MessageHeader, using default ConfigurationGroup parameter values for Action
and CPAId  ( Dummy Action, basic CPA )→
 <eb:MessageHeader>

  </eb:MessageHeader>
  </soap:Header>
  </soap:Envelope>
  </mime:MessageContainer>
  </mime:Message>
  </ebTest:MessageDeclaration>
  </ebTest:PutMessage>
  </ebTest:TestStep>
 <ebTest:TestStep >
 <ebTest:GetMessage ebTest:description="Correlate returned message">
<!—Filter returned messages from MessageStore, using RefToMessageId to find the desired
message →
  <ebTest:Filter>eb:CPAId=$CPAId and eb:Conversationid=$ConversationId and
eb:Action='Mute'</ebTest:Filter>
 <ebTest:TestAssertion ebTest:description="Verify that an ebXML message is returned">
 <ebTest:VerifyContent>/mime:Message[mime:MessageContainer[1]/soap:Envelope/soap:Header/
eb:MessageHeader]</ebTest:VerifyContent>
  </ebTest:TestAssertion>
  </ebTest:GetMessage>
  </ebTest:TestStep>
  </ebTest:TestCase>
<!—Define Test Case,referencing corresponding Functional Requirement in Interoperability
Test Requirements XML document →
```

```
 <ebTest:TestCase ebTest:requirementReferenceId="funreq_id_1.2"
ebTest:id="urn:TestCase:id:1.2" ebTest:description="Basic asyncronous exchange with one
payload">
 <ebTest:TestStep >
 <ebTest:PutMessage ebTest:description="Send basic message header">
 <ebTest:MessageDeclaration>
 <mime:Message>
 <mime:MessageContainer>
 <soap:Envelope>
 <soap:Header>
<!—Override "default" Action in MessageHeader declaration →
 <eb:MessageHeader>
  <eb:Action>Reflector</eb:Action>
  </eb:MessageHeader>
  </soap:Header>
 <soap:Body>
<!—Provide a Manifest declaration to include a reference to the payload →
 <eb:Manifest>
  <eb:Reference xlink:href="cid:payload_1" />
  </eb:Manifest>
  </soap:Body>
  </soap:Envelope>
  </mime:MessageContainer>
  </mime:Message>
  </ebTest:MessageDeclaration>
 <ebTest:SetPayload ebTest:description="Add content-id and payload to mime message">
  <ebTest:Content-ID>cid:payload_1</ebTest:Content-ID>
  <ebTest:MessageRef>payload_1</ebTest:MessageRef>
  </ebTest:SetPayload>
  </ebTest:PutMessage>
  </ebTest:TestStep>
 <ebTest:TestStep >
 <ebTest:GetMessage ebTest:description="Correlate returned messages">
<!—Filter returned messages from MessageStore, using RefToMessageId and Action name to
find the desired message →
  <ebTest:Filter>eb:CPAId='cpa_basic' and eb:Conversationid=$ConversationId and
eb:Action='Mute'</ebTest:Filter>
 <ebTest:TestAssertion ebTest:description="Check for returned payload">
 <ebTest:VerifyContent>/mime:Message[mime:MessageContainer[1]/soap:Body/eb:Manifest/eb:R
eference[@xlink:href='cid:payload_1']]</ebTest:VerifyContent>
  </ebTest:TestAssertion>
 <ebTest:GetPayload ebTest:description="Find payload in message">
  <ebTest:Content-ID>cid:payload_1</ebTest:Content-ID>
 <ebTest:TestAssertion ebTest:description="Verify returned payload contents">
  <ebTest:VerifyContent ebTest:verifyMethod="xpath" />
  </ebTest:TestAssertion>
  </ebTest:GetPayload>
  </ebTest:GetMessage>
  </ebTest:TestStep>
  </ebTest:TestCase>
<!—Define Test Case,referencing corresponding Functional Requirement in Interoperability
Test Requirements XML document →
 <ebTest:TestCase ebTest:requirementReferenceId="funreq_id_1.3"
ebTest:id="urn:TestCase:id:1.3" ebTest:description="Basic exchange with three payloads">
 <ebTest:TestStep >
 <ebTest:PutMessage ebTest:description="Send basic message header">
 <ebTest:MessageDeclaration>
 <mime:Message>
 <mime:MessageContainer>
 <soap:Envelope>
 <soap:Header>
<!—Override default value of Action in MessageHeader →
 <eb:MessageHeader>
  <eb:Action>Reflector</eb:Action>
  </eb:MessageHeader>
  </soap:Header>
 <soap:Body>
<!—Declare Manifest element with 3 payload references →
 <eb:Manifest>
  <eb:Reference xlink:href="cid:payload_1" />
```

```
  <eb:Reference xlink:href="cid:payload_2" />
  <eb:Reference xlink:href="cid:payload_3" />
  </eb:Manifest>
  </soap:Body>
  </soap:Envelope>
  </mime:MessageContainer>
  </mime:Message>
  </ebTest:MessageDeclaration>
 <ebTest:SetPayload ebTest:description="Add content-id and payload to mime message">
  <ebTest:Content-ID>cid:payload_1</ebTest:Content-ID>
  <ebTest:MessageRef>payload_1</ebTest:MessageRef>
  </ebTest:SetPayload>
 <ebTest:SetPayload ebTest:description="Add content-id and payload to mime message">
  <ebTest:Content-ID>cid:payload_2</ebTest:Content-ID>
  <ebTest:MessageRef>payload_2</ebTest:MessageRef>
  </ebTest:SetPayload>
 <ebTest:SetPayload ebTest:description="Add content-id and payload to mime message">
  <ebTest:Content-ID>payload_3</ebTest:Content-ID>
  <ebTest:MessageRef>payload_3</ebTest:MessageRef>
  </ebTest:SetPayload>
  </ebTest:PutMessage>
  </ebTest:TestStep>
 <ebTest:TestStep >
 <ebTest:GetMessage ebTest:description="Correlate returned messages">
  <ebTest:Filter>eb:CPAId='cpa_basic' and eb:Conversationid=$ConversationId and
eb:Action='Mute'</ebTest:Filter>
 <ebTest:TestAssertion ebTest:description="Check for returned payloads reference in
Manifest">
 <ebTest:VerifyContent>/mime:Message[mime:MessageContainer[1]/soap:Body/eb:Manifest[eb:R
eference[@xlink:href='cid:payload_1'] and eb:Reference[@xlink:href='cid:payload_2'] and
eb:Reference[@xlink:href='cid:payload_3']]</ebTest:VerifyContent>
  </ebTest:TestAssertion>
 <ebTest:GetPayload ebTest:description="Find payload in message">
  <ebTest:Content-ID>cid:payload_1</ebTest:Content-ID>
 <ebTest:TestAssertion ebTest:description="Verify returned payload contents">
  <ebTest:VerifyContent />
  </ebTest:TestAssertion>
  </ebTest:GetPayload>
 <ebTest:GetPayload ebTest:description="Find payload in message">
  <ebTest:Content-ID>cid:payload_2</ebTest:Content-ID>
 <ebTest:TestAssertion ebTest:description="Verify returned payload contents">
  <ebTest:VerifyContent />
  </ebTest:TestAssertion>
  </ebTest:GetPayload>
 <ebTest:GetPayload ebTest:description="Find payload in message">
  <ebTest:Content-ID>cid:payload_3</ebTest:Content-ID>
 <ebTest:TestAssertion ebTest:description="Verify returned payload contents">
  <ebTest:VerifyContent />
  </ebTest:TestAssertion>
  </ebTest:GetPayload>
  </ebTest:GetMessage>
  </ebTest:TestStep>
  </ebTest:TestCase>
<!—Define Test Case,referencing corresponding Functional Requirement in Interoperability
Test Requirements XML document →
 <ebTest:TestCase ebTest:requirementReferenceId="funreq_id_1.4"
ebTest:id="urn:TestCase:id:1.4" ebTest:description="Basic exchange with Error Message">
 <ebTest:TestStep >
 <ebTest:PutMessage ebTest:description="MessageHeader mustUnderstand set to 'true'">
 <ebTest:MessageDeclaration>
 <mime:Message>
 <mime:MessageContainer>
 <soap:Envelope>
 <soap:Header>
 <eb:MessageHeader>
  <eb:Action>Dummy</eb:Action>
  <eb:ExtensionElement soap:mustUnderstand="true" />
  </eb:MessageHeader>
  </soap:Header>
  </soap:Envelope>
  </mime:MessageContainer>
```

```
  </mime:Message>
  </ebTest:MessageDeclaration>
  </ebTest:PutMessage>
  </ebTest:TestStep>
 <ebTest:TestStep >
 <ebTest:GetMessage ebTest:getMultiple="true" ebTest:description="Correlate returned
messages">
  <ebTest:Filter>eb:CPAId='cpa_basic' and eb:Conversationid=$ConversationId and
eb:ErrorList</ebTest:Filter>
 <ebTest:TestAssertion ebTest:description="Test if Error is generated">
 <ebTest:VerifyContent>mime:Message[mime:MessageContainer[1]/soap:Envelope/soap:Body/soa
p:Fault/soap:Code[soap:Value='MustUnderstand']]</ebTest:VerifyContent>
  </ebTest:TestAssertion>
  </ebTest:GetMessage>
  </ebTest:TestStep>
  </ebTest:TestCase>
</ebTest:TestSuite>
```

Appendix A (Normative) The ebXML Test Profile Schema

The OASIS ebXML Implementation and Interoperability Committee has provided a version of the ebXML Test Profile schema using the schema vocabulary that conforms to the W3C XML Schema Recommendation specification [XMLSchema].

```
<?xml version="1.0" encoding="UTF-8" ?>
 <!--
Generated by XML Authority. Conforms to w3c http://www.w3.org/2001/XMLSchema
  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.oasis-
open.org/tc/ebxml-iic/test-profile" xmlns:tns="http://www.oasis-open.org/tc/ebxml-
iic/test-profile">
 <!--
 $Id: TestProfile.xsd,v 1.2 2002/07/02 15:28:27 matt Exp $
  -->
<element name="TestProfile">
<complexType>
<sequence>
  <element ref="tns:Dependency" minOccurs="0" maxOccurs="unbounded" />
  <element ref="tns:TestRequirementRef" maxOccurs="unbounded" />
  </sequence>
  <attribute name="requirementsLocation" use="required" type="anyURI" />
  <attribute name="name" use="required" type="string" />
  <attribute name="description" use="required" type="string" />
  </complexType>
  </element>
<element name="Dependency">
<complexType>
  <attribute name="name" use="required" type="string" />
  <attribute name="profileRef" use="required" type="anyURI" />
  </complexType>
  </element>
<element name="TestRequirementRef">
 <!--
 To overide the conformance type of the underlying requirement ...
  -->
<complexType>
<sequence>
  <element name="Comment" type="string" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attribute name="id" use="required" type="string" />
  <attribute name="requirementType" use="optional" type="tns:requirement.type" />
  </complexType>
  </element>
<simpleType name="requirement.type">
<restriction base="string">
  <enumeration value="required" />
  <enumeration value="strongly recommended" />
  <enumeration value="recommended" />
  <enumeration value="optional" />
  </restriction>
  </simpleType>
  </schema>
```

Appendix B (Normative) The ebXML Test Requirements Schema

The OASIS ebXML Implementation and Interoperability Committee has provided a version of the ebXML Test Requirements schema using the schema vocabulary that conforms to the W3C XML Schema Recommendation specification [XMLSchema].

```
<?xml version="1.0" encoding="UTF-8" ?>
 <!--
Generated by XML Authority. Conforms to w3c http://www.w3.org/2001/XMLSchema
  -->
<schema targetNamespace="http://www.oasis-open.org/tc/ebxml-iic/tests"
xmlns:mime="http://www.oasis-open.org/tc/ebxml-iic/tests/mime"
xmlns:ds="http://www.oasis-open.org/tc/ebxml-iic/tests/xmldsig"
xmlns:ebTest="http://www.oasis-open.org/tc/ebxml-iic/tests"
xmlns="http://www.w3.org/2001/XMLSchema" elementFormDefault="unqualified"
attributeFormDefault="unqualified" version="1.0">
  <import namespace="http://www.oasis-open.org/tc/ebxml-iic/tests/xmldsig"
schemaLocation="xmldsig.xsd" />
  <import namespace="http://www.oasis-open.org/tc/ebxml-iic/tests/mime"
schemaLocation="mime.xsd" />
 <!--
 edited with XML Spy v4.3 U (http://www.xmlspy.com) by Michael Kass (NIST)
  -->
 <!--
<import namespace="http://www.oasis-open.org/tc/ebxml-iic/tests/xmldsig"
schemaLocation="xmldsig.xsd"/>
  -->
 <!--
<import namespace="http://www.oasis-open.org/tc/ebxml-iic/tests/soap"
schemaLocation="soap.xsd"/>
  -->
<element name="TestSuite">
<complexType>
<sequence>
  <element ref="ebTest:MetaData" />
  <element ref="ebTest:ConfigurationGroup" maxOccurs="unbounded" />
  <element ref="ebTest:MessagePayload" minOccurs="0" maxOccurs="unbounded" />
<choice maxOccurs="unbounded">
  <element ref="ebTest:ConfigurationGroup" minOccurs="0" />
  <element ref="ebTest:TestCase" maxOccurs="unbounded" />
  </choice>
  </sequence>
  <attribute name="configurationGroupRef" type="IDREF" use="required" />
  </complexType>
  </element>
<element name="MetaData">
<complexType>
<sequence>
  <element ref="ebTest:Description" />
  <element ref="ebTest:Version" />
  <element ref="ebTest:Maintainer" />
  <element ref="ebTest:Location" />
  <element ref="ebTest:PublishDate" />
  <element ref="ebTest:Status" />
  </sequence>
  </complexType>
  </element>
  <element name="Description" type="ebTest:non-empty-string" />
  <element name="Version" type="ebTest:non-empty-string" />
  <element name="Maintainer" type="ebTest:non-empty-string" />
  <element name="Location" type="anyURI" />
```

```
  <element name="PublishDate" type="ebTest:non-empty-string" />
  <element name="Status" type="ebTest:non-empty-string" />
<element name="TestCase">
<complexType>
<sequence>
  <element ref="ebTest:TestStep" maxOccurs="unbounded" />
  </sequence>
  <attribute name="id" type="string" use="required" />
  <attribute name="description" type="string" use="required" />
  <attribute name="name" type="string" use="optional" />
  <attribute name="author" type="string" use="optional" />
  <attribute name="version" type="string" use="optional" />
  <attribute name="requirementReferenceId" type="anyURI" use="required" />
  <attribute name="configurationGroupRef" type="IDREF" use="optional" />
  </complexType>
  </element>
<element name="TestStep">
<complexType>
<choice>
  <element ref="ebTest:PutMessage" />
  <element ref="ebTest:GetMessage" />
  </choice>
  <attribute name="description" type="string" use="optional" />
  <attribute name="configurationGroupRef" type="IDREF" use="optional" />
  <attribute name="repeatTimes" type="integer" default="1" />
  <attribute name="testStepContext" type="integer" use="optional" />
  <attribute name="stepDelay" type="integer" use="optional" />
  </complexType>
  </element>
<element name="MessageExpression">
<complexType>
<sequence>
  <element ref="ebTest:ErrorMessage" />
  </sequence>
  </complexType>
  </element>
  <element name="ErrorMessage" type="ebTest:non-empty-string" />
<element name="PutMessage">
<complexType>
<sequence>
  <element ref="ebTest:MessageDeclaration" />
  <element ref="ebTest:SetPayload" minOccurs="0" maxOccurs="unbounded" />
<element name="DSign" minOccurs="0">
<complexType>
<sequence>
  <element ref="ds:Signature" maxOccurs="unbounded" />
  </sequence>
  </complexType>
  </element>
  <element ref="ebTest:SetParameter" minOccurs="0" />
  </sequence>
  <attribute name="description" type="string" use="required" />
  <attribute name="clearMessageStore" type="boolean" default="false" />
  </complexType>
  </element>
<element name="GetPayload">
<complexType>
<sequence>
<choice>
  <element ref="ebTest:Content-ID" />
  <element ref="ebTest:Content-Location" />
  <element ref="ebTest:Index" />
  </choice>
<choice minOccurs="0" maxOccurs="unbounded">
  <element ref="ebTest:TestPreCondition" />
  <element ref="ebTest:TestAssertion" />
  </choice>
  <element ref="ebTest:SetParameter" minOccurs="0" />
  </sequence>
  <attribute name="description" type="string" use="required" />
  </complexType>
```

```
  </element>
<element name="GetMessage">
<complexType>
<sequence>
  <element ref="ebTest:Filter" />
  <element ref="ebTest:TestPreCondition" minOccurs="0" maxOccurs="unbounded" />
  <element ref="ebTest:TestAssertion" minOccurs="0" maxOccurs="unbounded" />
  <element ref="ebTest:SetParameter" minOccurs="0" />
  <element ref="ebTest:GetPayload" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attribute name="description" type="string" use="required" />
  <attribute name="getMultiple" type="boolean" default="false" />
  <attribute name="testStepContext" type="integer" use="optional" />
  </complexType>
  </element>
  <element name="Filter" type="ebTest:non-empty-string" />
<element name="SetPayload">
<complexType>
<sequence>
<choice>
  <element ref="ebTest:Content-ID" />
  <element ref="ebTest:Content-Location" />
  </choice>
<choice>
  <element ref="ebTest:FileURI" />
  <element ref="ebTest:PayloadRef" />
  </choice>
<sequence minOccurs="0" maxOccurs="unbounded">
  <element ref="ebTest:MimeHeader" />
  <element ref="ebTest:MimeHeaderValue" />
  </sequence>
  <element ref="ebTest:SetParameter" minOccurs="0" />
  </sequence>
  <attribute name="description" type="string" use="required" />
  </complexType>
  </element>
<element name="TestPreCondition">
<complexType>
<choice>
  <element ref="ebTest:VerifyContent" />
  <element ref="ebTest:ValidateContent" />
  </choice>
  <attribute name="description" type="string" use="required" />
  </complexType>
  </element>
<element name="TestAssertion">
<complexType>
<choice>
  <element ref="ebTest:VerifyContent" />
  <element ref="ebTest:ValidateContent" />
  </choice>
  <attribute name="description" type="string" use="required" />
  <attribute name="requirement" type="ebTest:requirement.type" default="required" />
  </complexType>
  </element>
<simpleType name="mimeHeader.type">
<restriction base="NMTOKEN">
  <enumeration value="MIMEMessageContent-Type" />
  <enumeration value="MIMEMessageStart" />
  <enumeration value="Content-Type" />
  <enumeration value="start" />
  <enumeration value="charset" />
  <enumeration value="type" />
  <enumeration value="wildcard" />
  </restriction>
  </simpleType>
<simpleType name="content.type">
<restriction base="NMTOKEN">
  <enumeration value="XML" />
  <enumeration value="date" />
  <enumeration value="URI" />
```

```
3123        <enumeration value="signature" />
3124        <enumeration value="signedAck" />
3125        </restriction>
3126        </simpleType>
3127      <simpleType name="method.type">
3128      <restriction base="NMTOKEN">
3129        <enumeration value="xpath" />
3130        <enumeration value="sha-1" />
3131        </restriction>
3132        </simpleType>
3133      <simpleType name="mode.type">
3134      <restriction base="NMTOKEN">
3135        <enumeration value="service" />
3136        <enumeration value="connection" />
3137        </restriction>
3138        </simpleType>
3139      <simpleType name="messageContext.type">
3140      <restriction base="NMTOKEN">
3141        <enumeration value="true" />
3142        <enumeration value="false" />
3143        </restriction>
3144        </simpleType>
3145      <simpleType name="requirement.type">
3146      <restriction base="NMTOKEN">
3147        <enumeration value="required" />
3148        <enumeration value="stronglyrecommended" />
3149        <enumeration value="recommended" />
3150        <enumeration value="optional" />
3151        </restriction>
3152        </simpleType>
3153      <simpleType name="non-empty-string">
3154      <restriction base="string">
3155        <minLength value="1" />
3156        </restriction>
3157        </simpleType>
3158      <simpleType name="configAction.type">
3159      <restriction base="NMTOKEN">
3160        <enumeration value="query" />
3161        <enumeration value="replace" />
3162        </restriction>
3163        </simpleType>
3164      <simpleType name="action.type">
3165      <restriction base="NMTOKEN">
3166        <enumeration value="reset" />
3167        <enumeration value="modify" />
3168        </restriction>
3169        </simpleType>
3170      <simpleType name="configItem.type">
3171      <restriction base="NOTATION">
3172        <enumeration value="parameter" />
3173        <enumeration value="namespace" />
3174        </restriction>
3175        </simpleType>
3176      <simpleType name="parameter.type">
3177      <restriction base="NMTOKEN">
3178        <enumeration value="xpath" />
3179        <enumeration value="string" />
3180        </restriction>
3181        </simpleType>
3182        <element name="MimeHeader" type="ebTest:mimeHeader.type" />
3183        <element name="MimeHeaderValue" type="ebTest:non-empty-string" />
3184        <element name="Content-Location" type="ebTest:non-empty-string" />
3185        <element name="Index" type="integer" />
3186        <element name="FileURI" type="anyURI" />
3187        <element name="PayloadRef" type="string" />
3188        <element name="Signature" type="base64Binary" />
3189        <element name="Content-ID" type="string" />
3190      <element name="MessageDeclaration">
3191      <complexType>
3192      <sequence>
3193        <element ref="mime:Message" />
```

```
  </sequence>
  </complexType>
  </element>
<element name="ValidateContent">
<complexType>
<simpleContent>
<extension base="ebTest:non-empty-string">
  <attribute name="contentType" type="ebTest:content.type" default="XML" />
  <attribute name="schemaLocation" type="anyURI" use="optional" />
  </extension>
  </simpleContent>
  </complexType>
  </element>
 <element name="VerifyContent">
 <complexType>
 <simpleContent>
 <extension base="string">
  <attribute name="verifyMethod" type="ebTest:method.type" default="xpath" />
  </extension>
  </simpleContent>
  </complexType>
  </element>
 <element name="MessagePayload">
 <complexType>
 <sequence>
  <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attribute name="id" type="ID" use="required" />
  </complexType>
  </element>
 <element name="ConfigurationGroup">
 <complexType>
 <sequence>
  <element ref="ebTest:CPAId" />
  <element ref="ebTest:Mode" />
  <element ref="ebTest:SenderParty" />
  <element ref="ebTest:ReceiverParty" />
  <element ref="ebTest:Service" />
  <element ref="ebTest:Action" />
  <element ref="ebTest:StepDelay" />
  <element ref="ebTest:ResponseURL" minOccurs="0" />
  <element ref="ebTest:NotifcationURL" minOccurs="0" />
 <element name="PayloadDigests" minOccurs="0">
 <complexType>
 <sequence>
 <element name="Payload" maxOccurs="unbounded">
 <complexType>
 <sequence>
  <element name="Href" type="anyURI" />
  <element name="Digest" type="base64Binary" />
  </sequence>
  </complexType>
  </element>
  </sequence>
  </complexType>
  </element>
 <element name="ConfigurationItem" minOccurs="0" maxOccurs="unbounded">
 <complexType>
 <sequence>
  <element name="Name" type="ebTest:non-empty-string" />
  <element name="Value" type="ebTest:non-empty-string" />
  <element name="Type" type="ebTest:configItem.type" />
  </sequence>
  </complexType>
  </element>
  </sequence>
  <attribute name="id" type="ID" use="required" />
  </complexType>
  </element>
  <element name="SenderParty" type="ebTest:non-empty-string" />
  <element name="ReceiverParty" type="ebTest:non-empty-string" />
```

```
 <element name="Service" type="anyURI" />
 <element name="Action" type="ebTest:non-empty-string" />
 <element name="StepDelay" type="integer" />
 <element name="ResponseURL" type="anyURI" />
 <element name="NotifcationURL" type="anyURI" />
 <element name="CPAId" type="ebTest:non-empty-string" />
 <element name="Mode" type="ebTest:non-empty-string" />
<element name="SetParameter">
<complexType>
<sequence>
 <element name="Name" type="string" />
 <element name="Value" type="string" />
 </sequence>
 <attribute name="parameterType" type="ebTest:parameter.type" use="required" />
 </complexType>
 </element>
 </schema>
```

Appendix C (Normative) The ebXML Test Suite Schema and Supporting Subschemas

The OASIS ebXML Implementation and Interoperability Committee has provided a version of the ebXML Test Requirements schema using the schema vocabulary that conforms to the W3C XML Schema Recommendation specification [XMLSchema].

```
<?xml version="1.0" encoding="UTF-8" ?>
 <!--
Generated by XML Authority. Conforms to w3c http://www.w3.org/2001/XMLSchema
  -->
 <schema targetNamespace="http://www.oasis-open.org/tc/ebxml-iic/tests"
xmlns:mime="http://www.oasis-open.org/tc/ebxml-iic/tests/mime"
xmlns:ds="http://www.oasis-open.org/tc/ebxml-iic/tests/xmldsig"
xmlns:ebTest="http://www.oasis-open.org/tc/ebxml-iic/tests"
xmlns="http://www.w3.org/2001/XMLSchema" elementFormDefault="unqualified"
attributeFormDefault="unqualified" version="1.0">
  <import namespace="http://www.oasis-open.org/tc/ebxml-iic/tests/xmldsig"
schemaLocation="xmldsig.xsd" />
  <import namespace="http://www.oasis-open.org/tc/ebxml-iic/tests/mime"
schemaLocation="mime.xsd" />
 <!--
 edited with XML Spy v4.3 U (http://www.xmlspy.com) by Michael Kass (NIST)
  -->
 <!--
<import namespace="http://www.oasis-open.org/tc/ebxml-iic/tests/xmldsig"
schemaLocation="xmldsig.xsd"/>
  -->
 <!--
<import namespace="http://www.oasis-open.org/tc/ebxml-iic/tests/soap"
schemaLocation="soap.xsd"/>
  -->
 <element name="TestSuite">
 <complexType>
 <sequence>
  <element ref="ebTest:MetaData" />
  <element ref="ebTest:ConfigurationGroup" maxOccurs="unbounded" />
  <element ref="ebTest:MessagePayload" minOccurs="0" maxOccurs="unbounded" />
 <choice maxOccurs="unbounded">
  <element ref="ebTest:ConfigurationGroup" minOccurs="0" />
  <element ref="ebTest:TestCase" maxOccurs="unbounded" />
  </choice>
  </sequence>
  <attribute name="configurationGroupRef" type="IDREF" use="required" />
  </complexType>
  </element>
 <element name="MetaData">
 <complexType>
 <sequence>
  <element ref="ebTest:Description" />
  <element ref="ebTest:Version" />
  <element ref="ebTest:Maintainer" />
  <element ref="ebTest:Location" />
  <element ref="ebTest:PublishDate" />
  <element ref="ebTest:Status" />
  </sequence>
  </complexType>
  </element>
  <element name="Description" type="ebTest:non-empty-string" />
  <element name="Version" type="ebTest:non-empty-string" />
  <element name="Maintainer" type="ebTest:non-empty-string" />
  <element name="Location" type="anyURI" />
  <element name="PublishDate" type="ebTest:non-empty-string" />
  <element name="Status" type="ebTest:non-empty-string" />
```

```
<element name="TestCase">
<complexType>
<sequence>
 <element ref="ebTest:TestStep" maxOccurs="unbounded" />
 </sequence>
 <attribute name="id" type="string" use="required" />
 <attribute name="description" type="string" use="required" />
 <attribute name="name" type="string" use="optional" />
 <attribute name="author" type="string" use="optional" />
 <attribute name="version" type="string" use="optional" />
 <attribute name="requirementReferenceId" type="anyURI" use="required" />
 <attribute name="configurationGroupRef" type="IDREF" use="optional" />
 </complexType>
 </element>
<element name="TestStep">
<complexType>
<choice>
 <element ref="ebTest:PutMessage" />
 <element ref="ebTest:GetMessage" />
 </choice>
 <attribute name="description" type="string" use="optional" />
 <attribute name="configurationGroupRef" type="IDREF" use="optional" />
 <attribute name="repeatTimes" type="integer" default="1" />
 <attribute name="testStepContext" type="integer" use="optional" />
 <attribute name="stepDelay" type="integer" use="optional" />
 </complexType>
 </element>
<element name="MessageExpression">
<complexType>
<sequence>
 <element ref="ebTest:ErrorMessage" />
 </sequence>
 </complexType>
 </element>
 <element name="ErrorMessage" type="ebTest:non-empty-string" />
<element name="PutMessage">
<complexType>
<sequence>
 <element ref="ebTest:MessageDeclaration" />
 <element ref="ebTest:SetPayload" minOccurs="0" maxOccurs="unbounded" />
<element name="DSign" minOccurs="0">
<complexType>
<sequence>
 <element ref="ds:Signature" maxOccurs="unbounded" />
 </sequence>
 </complexType>
 </element>
 <element ref="ebTest:SetParameter" minOccurs="0" />
 </sequence>
 <attribute name="description" type="string" use="required" />
 <attribute name="clearMessageStore" type="boolean" default="false" />
 </complexType>
 </element>
<element name="GetPayload">
<complexType>
<sequence>
<choice>
 <element ref="ebTest:Content-ID" />
 <element ref="ebTest:Content-Location" />
 <element ref="ebTest:Index" />
 </choice>
<choice minOccurs="0" maxOccurs="unbounded">
 <element ref="ebTest:TestPreCondition" />
 <element ref="ebTest:TestAssertion" />
 </choice>
 <element ref="ebTest:SetParameter" minOccurs="0" />
 </sequence>
 <attribute name="description" type="string" use="required" />
 </complexType>
 </element>
<element name="GetMessage">
```

```
<complexType>
<sequence>
 <element ref="ebTest:Filter" />
 <element ref="ebTest:TestPreCondition" minOccurs="0" maxOccurs="unbounded" />
 <element ref="ebTest:TestAssertion" minOccurs="0" maxOccurs="unbounded" />
 <element ref="ebTest:SetParameter" minOccurs="0" />
 <element ref="ebTest:GetPayload" minOccurs="0" maxOccurs="unbounded" />
 </sequence>
 <attribute name="description" type="string" use="required" />
 <attribute name="getMultiple" type="boolean" default="false" />
 <attribute name="testStepContext" type="integer" use="optional" />
 </complexType>
 </element>
 <element name="Filter" type="ebTest:non-empty-string" />
<element name="SetPayload">
<complexType>
<sequence>
<choice>
 <element ref="ebTest:Content-ID" />
 <element ref="ebTest:Content-Location" />
 </choice>
<choice>
 <element ref="ebTest:FileURI" />
 <element ref="ebTest:PayloadRef" />
 </choice>
<sequence minOccurs="0" maxOccurs="unbounded">
 <element ref="ebTest:MimeHeader" />
 <element ref="ebTest:MimeHeaderValue" />
 </sequence>
 <element ref="ebTest:SetParameter" minOccurs="0" />
 </sequence>
 <attribute name="description" type="string" use="required" />
 </complexType>
 </element>
<element name="TestPreCondition">
<complexType>
<choice>
 <element ref="ebTest:VerifyContent" />
 <element ref="ebTest:ValidateContent" />
 </choice>
 <attribute name="description" type="string" use="required" />
 </complexType>
 </element>
<element name="TestAssertion">
<complexType>
<choice>
 <element ref="ebTest:VerifyContent" />
 <element ref="ebTest:ValidateContent" />
 </choice>
 <attribute name="description" type="string" use="required" />
 <attribute name="requirement" type="ebTest:requirement.type" default="required" />
 </complexType>
 </element>
<simpleType name="mimeHeader.type">
<restriction base="NMTOKEN">
 <enumeration value="MIMEMessageContent-Type" />
 <enumeration value="MIMEMessageStart" />
 <enumeration value="Content-Type" />
 <enumeration value="start" />
 <enumeration value="charset" />
 <enumeration value="type" />
 <enumeration value="wildcard" />
 </restriction>
 </simpleType>
<simpleType name="content.type">
<restriction base="NMTOKEN">
 <enumeration value="XML" />
 <enumeration value="date" />
 <enumeration value="URI" />
 <enumeration value="signature" />
 <enumeration value="signedAck" />
```

```
 </restriction>
 </simpleType>
<simpleType name="method.type">
<restriction base="NMTOKEN">
 <enumeration value="xpath" />
 <enumeration value="sha-1" />
 </restriction>
 </simpleType>
<simpleType name="mode.type">
<restriction base="NMTOKEN">
 <enumeration value="service" />
 <enumeration value="connection" />
 </restriction>
 </simpleType>
<simpleType name="messageContext.type">
<restriction base="NMTOKEN">
 <enumeration value="true" />
 <enumeration value="false" />
 </restriction>
 </simpleType>
<simpleType name="requirement.type">
<restriction base="NMTOKEN">
 <enumeration value="required" />
 <enumeration value="stronglyrecommended" />
 <enumeration value="recommended" />
 <enumeration value="optional" />
 </restriction>
 </simpleType>
<simpleType name="non-empty-string">
<restriction base="string">
 <minLength value="1" />
 </restriction>
 </simpleType>
<simpleType name="configAction.type">
<restriction base="NMTOKEN">
 <enumeration value="query" />
 <enumeration value="replace" />
 </restriction>
 </simpleType>
<simpleType name="action.type">
<restriction base="NMTOKEN">
 <enumeration value="reset" />
 <enumeration value="modify" />
 </restriction>
 </simpleType>
<simpleType name="configItem.type">
<restriction base="NOTATION">
 <enumeration value="parameter" />
 <enumeration value="namespace" />
 </restriction>
 </simpleType>
<simpleType name="parameter.type">
<restriction base="NMTOKEN">
 <enumeration value="xpath" />
 <enumeration value="string" />
 </restriction>
 </simpleType>
 <element name="MimeHeader" type="ebTest:mimeHeader.type" />
 <element name="MimeHeaderValue" type="ebTest:non-empty-string" />
 <element name="Content-Location" type="ebTest:non-empty-string" />
 <element name="Index" type="integer" />
 <element name="FileURI" type="anyURI" />
 <element name="PayloadRef" type="string" />
 <element name="Signature" type="base64Binary" />
 <element name="Content-ID" type="string" />
<element name="MessageDeclaration">
<complexType>
<sequence>
 <element ref="mime:Message" />
 </sequence>
 </complexType>
```

```
 </element>
<element name="ValidateContent">
<complexType>
<simpleContent>
<extension base="ebTest:non-empty-string">
 <attribute name="contentType" type="ebTest:content.type" default="XML" />
 <attribute name="schemaLocation" type="anyURI" use="optional" />
 </extension>
 </simpleContent>
 </complexType>
 </element>
<element name="VerifyContent">
<complexType>
<simpleContent>
<extension base="string">
 <attribute name="verifyMethod" type="ebTest:method.type" default="xpath" />
 </extension>
 </simpleContent>
 </complexType>
 </element>
<element name="MessagePayload">
<complexType>
<sequence>
 <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
 </sequence>
 <attribute name="id" type="ID" use="required" />
 </complexType>
 </element>
<element name="ConfigurationGroup">
<complexType>
<sequence>
 <element ref="ebTest:CPAId" />
 <element ref="ebTest:Mode" />
 <element ref="ebTest:SenderParty" />
 <element ref="ebTest:ReceiverParty" />
 <element ref="ebTest:Service" />
 <element ref="ebTest:Action" />
 <element ref="ebTest:StepDelay" />
 <element ref="ebTest:ResponseURL" minOccurs="0" />
 <element ref="ebTest:NotifcationURL" minOccurs="0" />
<element name="PayloadDigests" minOccurs="0">
<complexType>
<sequence>
<element name="Payload" maxOccurs="unbounded">
<complexType>
<sequence>
 <element name="Href" type="anyURI" />
 <element name="Digest" type="base64Binary" />
 </sequence>
 </complexType>
 </element>
 </sequence>
 </complexType>
 </element>
<element name="ConfigurationItem" minOccurs="0" maxOccurs="unbounded">
<complexType>
<sequence>
 <element name="Name" type="ebTest:non-empty-string" />
 <element name="Value" type="ebTest:non-empty-string" />
 <element name="Type" type="ebTest:configItem.type" />
 </sequence>
 </complexType>
 </element>
 </sequence>
 <attribute name="id" type="ID" use="required" />
 </complexType>
 </element>
 <element name="SenderParty" type="ebTest:non-empty-string" />
 <element name="ReceiverParty" type="ebTest:non-empty-string" />
 <element name="Service" type="anyURI" />
 <element name="Action" type="ebTest:non-empty-string" />
```

```
  <element name="StepDelay" type="integer" />
  <element name="ResponseURL" type="anyURI" />
  <element name="NotifcationURL" type="anyURI" />
  <element name="CPAId" type="ebTest:non-empty-string" />
  <element name="Mode" type="ebTest:non-empty-string" />
 <element name="SetParameter">
 <complexType>
 <sequence>
  <element name="Name" type="string" />
  <element name="Value" type="string" />
  </sequence>
  <attribute name="parameterType" type="ebTest:parameter.type" use="required" />
  </complexType>
  </element>
  </schema>
```

SOAP Message Declaration Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
 <!--
Generated by XML Authority. Conforms to w3c http://www.w3.org/2001/XMLSchema
  -->
 <schema targetNamespace="http://www.oasis-open.org/tc/ebxml-iic/tests/soap"
xmlns:eb="http://www.oasis-open.org/tc/ebxml-iic/tests/eb"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://www.oasis-
open.org/tc/ebxml-iic/tests/soap" xmlns="http://www.w3.org/2001/XMLSchema">
  <import namespace="http://www.oasis-open.org/tc/ebxml-iic/tests/eb"
schemaLocation="eb.xsd" />
 <attributeGroup name="encodingStyle">
  <attribute name="encodingStyle" type="tns:encodingStyle" />
  </attributeGroup>
 <!--
 Schema for the SOAP/1.1 envelope

     This schema has been produced using W3C's SOAP Version 1.2 schema
     found at:

     http://www.w3.org/2001/06/soap-envelope

     Copyright 2001 Martin Gudgin, Developmentor.

     Changes made are the following:
     - reverted namespace to http://schemas.xmlsoap.org/soap/envelope/
     - reverted mustUnderstand to only allow 0 and 1 as lexical values


     Copyright 2003 OASIS

     Changes made are the following:
     - SOAP Header and Body element content models constrained to include ebXML content


     Original copyright:

     Copyright 2001 W3C (Massachusetts Institute of Technology,
     Institut National de Recherche en Informatique et en Automatique,
     Keio University). All Rights Reserved.
     http://www.w3.org/Consortium/Legal/

     This document is governed by the W3C Software License [1] as
     described in the FAQ [2].

     [1] http://www.w3.org/Consortium/Legal/copyright-software-19980720
```

```
     [2] http://www.w3.org/Consortium/Legal/IPR-FAQ-20000620.html#DTD
  -->
 <!--
 Envelope, header and body
  -->
  <element name="Envelope" type="tns:Envelope" />
 <complexType name="Envelope">
 <sequence>
  <element ref="tns:Header" />
  <element ref="tns:Body" />
  <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <anyAttribute namespace="##other" processContents="lax" />
  </complexType>
 <group name="optionElements">
 <all minOccurs="0">
  <element ref="eb:SyncReply" minOccurs="0"/>
  <element ref="eb:MessageOrder" minOccurs="0"/>
  <element ref="eb:AckRequested" minOccurs="0"/>
  <element ref="eb:Acknowledgment" minOccurs="0"/>
  <element ref="eb:ErrorList" minOccurs="0"/>
  </all>
  </group>
 <element name="Header">
 <complexType>
 <sequence>
  <element ref="eb:MessageHeader" />
  <group ref="tns:optionElements" />
  </sequence>
  </complexType>
  </element>
 <complexType name="Header">
 <sequence>
  <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <anyAttribute namespace="##other" processContents="lax" />
  </complexType>
 <element name="Body">
 <complexType>
 <choice minOccurs="0">
  <element ref="eb:Manifest" />
  <element ref="eb:StatusRequest" />
  <element ref="eb:StatusResponse" />
  </choice>
  </complexType>
  </element>
 <complexType name="Body">
 <annotation>
  <documentation>Prose in the spec does not specify that attributes are allowed on the
Body element</documentation>
  </annotation>
 <sequence>
  <any namespace="##any" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <anyAttribute namespace="##any" processContents="lax" />
  </complexType>
 <!--
 Global Attributes.  The following attributes are intended to be usable via qualified
attribute names on any complex type referencing them.
  -->
 <attribute name="mustUnderstand" default="0">
 <simpleType>
 <restriction base="boolean">
  <pattern value="0|1" />
  </restriction>
  </simpleType>
  </attribute>
  <attribute name="actor" type="anyURI" />
 <simpleType name="encodingStyle">
 <annotation>
```

```
  <documentation>'encodingStyle' indicates any canonicalization conventions followed in
the contents of the containing element. For example, the value
'http://schemas.xmlsoap.org/soap/encoding/' indicates the pattern described in SOAP
specification</documentation>
  </annotation>
  <list itemType="anyURI" />
  </simpleType>
 <complexType name="Fault" final="extension">
 <annotation>
  <documentation>Fault reporting structure</documentation>
  </annotation>
 <sequence>
  <element name="faultcode" type="QName" />
  <element name="faultstring" type="string" />
  <element name="faultactor" type="anyURI" minOccurs="0" />
  <element name="detail" type="tns:detail" minOccurs="0" />
  </sequence>
  </complexType>
 <complexType name="detail">
 <sequence>
  <any namespace="##any" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <anyAttribute namespace="##any" processContents="lax" />
  </complexType>
  </schema>
```

ebXML Message Declaration Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
Generated by XML Authority. Conforms to w3c http://www.w3.org/2001/XMLSchema
  -->
 <schema targetNamespace="http://www.oasis-open.org/tc/ebxml-iic/tests/eb"
xmlns:soap="http://www.oasis-open.org/tc/ebxml-iic/tests/soap"
xmlns:ds="http://www.oasis-open.org/tc/ebxml-iic/tests/xmldsig"
xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:tns="http://www.oasis-
open.org/tc/ebxml-iic/tests/eb" xmlns="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="qualified" version="1.0">
  <import namespace="http://www.w3.org/1999/xlink" schemaLocation="http://www.oasis-
open.org/committees/ebxml-msg/schema/xlink.xsd" />
  <import namespace="http://www.oasis-open.org/tc/ebxml-iic/tests/xmldsig"
schemaLocation="xmldsig.xsd" />
  <import namespace="http://www.oasis-open.org/tc/ebxml-iic/tests/soap"
schemaLocation="soap.xsd" />
  <import namespace="http://www.w3.org/XML/1998/namespace"
schemaLocation="http://www.oasis-open.org/committees/ebxml-msg/schema/xml lang.xsd" />
 <attributeGroup name="bodyExtension.grp">
  <attribute ref="tns:id" />
  <attribute ref="tns:version" use="optional" />
  </attributeGroup>
 <attributeGroup name="headerExtension.grp">
  <attribute ref="tns:id" />
  <attribute ref="tns:version" use="optional" />
  <attribute ref="soap:mustUnderstand" use="optional" />
  </attributeGroup>
 <!--
 MANIFEST, for use in soap:Body element
  -->
 <element name="Manifest">
 <complexType>
 <sequence>
  <element ref="tns:Reference" maxOccurs="unbounded" />
```

```
 <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
 </sequence>
 <attributeGroup ref="tns:bodyExtension.grp" />
 </complexType>
 </element>
<element name="Reference">
<complexType>
<sequence>
 <element ref="tns:Schema" minOccurs="0" maxOccurs="unbounded" />
 <element ref="tns:Description" minOccurs="0" maxOccurs="unbounded" />
<choice>
 <element ref="tns:FileName" />
 <element ref="tns:MessageRef" />
 <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
 </choice>
 </sequence>
 <attribute ref="tns:id" />
 <attribute ref="xlink:type" fixed="simple" />
 <attribute ref="xlink:href" use="required" />
 <attribute ref="xlink:role" />
 <attribute name="contentId" type="string" use="optional" />
 <attribute name="contentType" type="string" use="optional" />
 <attribute name="contentLocation" type="anyURI" use="optional" />
 </complexType>
 </element>
<element name="Schema">
<complexType>
 <attribute name="location" type="anyURI" use="required" />
 <attribute name="version" type="tns:non-empty-string" />
 </complexType>
 </element>
<!--
MESSAGEHEADER, for use in soap:Header element
 -->
<element name="MessageHeader">
<complexType>
<sequence>
 <element ref="tns:From" minOccurs="0" />
 <element ref="tns:To" minOccurs="0" />
 <element ref="tns:CPAId" minOccurs="0" />
 <element ref="tns:ConversationId" minOccurs="0" />
 <element ref="tns:Service" minOccurs="0" />
 <element ref="tns:Action" minOccurs="0" />
 <element ref="tns:MessageData" minOccurs="0" />
 <element ref="tns:DuplicateElimination" minOccurs="0" />
 <element ref="tns:Description" minOccurs="0" maxOccurs="unbounded" />
 <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
 </sequence>
 <attributeGroup ref="tns:headerExtension.grp" />
 </complexType>
 </element>
 <element name="CPAId" type="tns:non-empty-string" />
 <element name="ConversationId" type="tns:non-empty-string" />
<element name="Service">
<complexType>
<simpleContent>
<extension base="tns:non-empty-string">
 <attribute name="type" type="tns:non-empty-string" />
 </extension>
 </simpleContent>
 </complexType>
 </element>
 <element name="Action" type="tns:non-empty-string" />
<element name="MessageData">
<complexType>
<sequence>
 <element ref="tns:MessageId" minOccurs="0" />
 <element ref="tns:Timestamp" minOccurs="0" />
 <element ref="tns:RefToMessageId" minOccurs="0" />
 <element ref="tns:TimeToLive" minOccurs="0" />
 </sequence>
```

```
 </complexType>
 </element>
 <element name="MessageId" type="tns:non-empty-string" />
 <element name="TimeToLive" type="dateTime" />
 <element name="DuplicateElimination" />
<!--
SYNC REPLY, for use in soap:Header element
 -->
<element name="SyncReply">
<complexType>
<sequence>
 <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
 </sequence>
 <attributeGroup ref="tns:headerExtension.grp" />
 <attribute ref="soap:actor" default="urn:oasis:names:tc:ebxml-msg:actor:toPartyMSH" />
 </complexType>
 </element>
<!--
MESSAGE ORDER, for use in soap:Header element
 -->
<element name="MessageOrder">
<complexType>
<sequence>
 <element ref="tns:SequenceNumber" />
 <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
 </sequence>
 <attributeGroup ref="tns:headerExtension.grp" />
 </complexType>
 </element>
 <element name="SequenceNumber" type="tns:sequenceNumber.type" />
<!--
ACK REQUESTED, for use in soap:Header element
 -->
<element name="AckRequested">
<complexType>
<sequence>
 <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
 </sequence>
 <attributeGroup ref="tns:headerExtension.grp" />
 <attribute ref="soap:actor" />
 <attribute name="signed" type="boolean" use="optional" />
 </complexType>
 </element>
<!--
ACKNOWLEDGMENT, for use in soap:Header element
 -->
<element name="Acknowledgment">
<complexType>
<sequence>
 <element ref="tns:Timestamp" minOccurs="0" />
 <element ref="tns:RefToMessageId" minOccurs="0" />
 <element ref="tns:From" minOccurs="0" />
 <element name="Reference" minOccurs="0" maxOccurs="unbounded" />
 <any namespace="##other" processContents="lax" minOccurs="0" />
 <element ref="ds:Reference" minOccurs="0" maxOccurs="unbounded" />
 </sequence>
 <attributeGroup ref="tns:headerExtension.grp" />
 <attribute ref="soap:actor" default="urn:oasis:names:tc:ebxml-msg:actor:toPartyMSH" />
 </complexType>
 </element>
<!--
ERROR LIST, for use in soap:Header element
 -->
<element name="ErrorList">
<complexType>
<sequence>
 <element ref="tns:Error" maxOccurs="unbounded" />
 <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
 </sequence>
 <attributeGroup ref="tns:headerExtension.grp" />
 <attribute name="highestSeverity" type="tns:severity.type" use="required" />
```

```
  </complexType>
  </element>
 <element name="Error">
 <complexType>
 <sequence>
  <element ref="tns:Description" minOccurs="0" />
  <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attribute ref="tns:id" />
  <attribute name="codeContext" type="anyURI" default="urn:oasis:names:tc:ebxml-
msg:service:errors" />
  <attribute name="errorCode" type="tns:non-empty-string" use="required" />
  <attribute name="severity" type="tns:severity.type" use="required" />
  <attribute name="location" type="tns:non-empty-string" />
  </complexType>
  </element>
 <!--
 STATUS RESPONSE, for use in soap:Body element
  -->
 <element name="StatusResponse">
 <complexType>
 <sequence>
  <element ref="tns:RefToMessageId" />
  <element ref="tns:Timestamp" minOccurs="0" />
  <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attributeGroup ref="tns:bodyExtension.grp" />
  <attribute name="messageStatus" type="tns:messageStatus.type" use="required" />
  </complexType>
  </element>
 <!--
 STATUS REQUEST, for use in soap:Body element
  -->
 <element name="StatusRequest">
 <complexType>
 <sequence>
  <element ref="tns:RefToMessageId" />
  <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attributeGroup ref="tns:bodyExtension.grp" />
  </complexType>
  </element>
 <!--
 COMMON TYPES
  -->
 <complexType name="sequenceNumber.type">
 <simpleContent>
 <extension base="positiveInteger">
  <attribute name="status" type="tns:status.type" default="Continue" />
  </extension>
  </simpleContent>
  </complexType>
 <simpleType name="status.type">
 <restriction base="NMTOKEN">
  <enumeration value="Reset" />
  <enumeration value="Continue" />
  </restriction>
  </simpleType>
 <simpleType name="messageStatus.type">
 <restriction base="NMTOKEN">
  <enumeration value="UnAuthorized" />
  <enumeration value="NotRecognized" />
  <enumeration value="Received" />
  <enumeration value="Processed" />
  <enumeration value="Forwarded" />
  </restriction>
  </simpleType>
 <simpleType name="non-empty-string">
 <restriction base="string">
  <minLength value="1" />
  </restriction>
```

```
 </simpleType>
<simpleType name="severity.type">
<restriction base="NMTOKEN">
 <enumeration value="Warning" />
 <enumeration value="Error" />
 </restriction>
 </simpleType>
<!--
COMMON ATTRIBUTES and ATTRIBUTE GROUPS
 -->
 <attribute name="id" type="ID" />
 <attribute name="version" type="tns:non-empty-string" />
<!--
COMMON ELEMENTS
 -->
<element name="PartyId">
<complexType>
<simpleContent>
<extension base="tns:non-empty-string">
 <attribute name="type" type="tns:non-empty-string" />
 </extension>
 </simpleContent>
 </complexType>
 </element>
<element name="To">
<complexType>
<sequence>
 <element ref="tns:PartyId" />
 <element name="Role" type="tns:non-empty-string" minOccurs="0" />
 </sequence>
 </complexType>
 </element>
<element name="From">
<complexType>
<sequence>
 <element ref="tns:PartyId" />
 <element name="Role" type="tns:non-empty-string" minOccurs="0" />
 </sequence>
 </complexType>
 </element>
<element name="Description">
<complexType>
<simpleContent>
<extension base="tns:non-empty-string">
 <attribute ref="xml:lang" use="required" />
 </extension>
 </simpleContent>
 </complexType>
 </element>
 <element name="RefToMessageId" type="tns:non-empty-string" />
 <element name="Timestamp" type="dateTime" />
 <element name="FileName" type="tns:non-empty-string" />
 <element name="MessageRef" type="tns:non-empty-string" />
 </schema>
```

XMLDSIG Declaration Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
 <!--
Generated by XML Authority. Conforms to w3c http://www.w3.org/2001/XMLSchema
  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.oasis-
open.org/tc/ebxml-iic/tests/xmldsig" xmlns:ds="http://www.oasis-open.org/tc/ebxml-
iic/tests/xmldsig" version="0.1" elementFormDefault="qualified"
attributeFormDefault="unqualified">
```

```
 <!--
 Basic Types Defined for Signatures
  -->
<simpleType name="CryptoBinary">
  <restriction base="base64Binary" />
  </simpleType>
<simpleType name="DigestValueType">
  <restriction base="base64Binary" />
  </simpleType>
<simpleType name="HMACOutputLengthType">
  <restriction base="integer" />
  </simpleType>
 <!--
 Start Signature
  -->
  <element name="Signature" type="ds:SignatureType" />
<complexType name="SignatureType">
<sequence>
  <element ref="ds:SignedInfo" />
  <element ref="ds:SignatureValue" minOccurs="0" />
  <element ref="ds:KeyInfo" minOccurs="0" />
  <element ref="ds:Object" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attribute name="Id" use="optional" type="ID" />
  </complexType>
  <element name="SignatureValue" type="ds:SignatureValueType" />
<complexType name="SignatureValueType">
<simpleContent>
<extension base="base64Binary">
  <attribute name="Id" use="optional" type="ID" />
  </extension>
  </simpleContent>
  </complexType>
 <!--
 Start SignedInfo
  -->
  <element name="SignedInfo" type="ds:SignedInfoType" />
<complexType name="SignedInfoType">
<sequence>
  <element ref="ds:CanonicalizationMethod" minOccurs="0" />
  <element ref="ds:SignatureMethod" minOccurs="0" />
  <element ref="ds:Reference" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attribute name="Id" use="optional" type="ID" />
  </complexType>
  <element name="CanonicalizationMethod" type="ds:CanonicalizationMethodType" />
<complexType name="CanonicalizationMethodType" mixed="true">
 <!--
 (0,unbounded) elements from (1,1) namespace
  -->
<sequence>
  <any namespace="##any" processContents="strict" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attribute name="Algorithm" use="required" type="anyURI" />
  </complexType>
  <element name="SignatureMethod" type="ds:SignatureMethodType" />
<complexType name="SignatureMethodType" mixed="true">
 <!--
 (0,unbounded) elements from (1,1) external namespace
  -->
<sequence>
  <element name="HMACOutputLength" type="ds:HMACOutputLengthType" minOccurs="0" />
  <any namespace="##other" processContents="strict" minOccurs="0" maxOccurs="unbounded"
/>
  </sequence>
  <attribute name="Algorithm" use="required" type="anyURI" />
  </complexType>
 <!--
 Start Reference
  -->
  <element name="Reference" type="ds:ReferenceType" />
```

```
<complexType name="ReferenceType">
<sequence>
  <element ref="ds:Transforms" minOccurs="0" />
  <element ref="ds:DigestMethod" />
  <element ref="ds:DigestValue" minOccurs="0" />
  </sequence>
  <attribute name="Id" use="optional" type="ID" />
  <attribute name="URI" use="optional" type="anyURI" />
  <attribute name="Type" use="optional" type="anyURI" />
  </complexType>
  <element name="Transforms" type="ds:TransformsType" />
<complexType name="TransformsType">
<sequence>
  <element ref="ds:Transform" maxOccurs="unbounded" />
  </sequence>
  </complexType>
  <element name="Transform" type="ds:TransformType" />
<complexType name="TransformType" mixed="true">
 <!--
 (1,1) elements from (0,unbounded) namespaces
  -->
<choice minOccurs="0" maxOccurs="unbounded">
  <any namespace="##other" processContents="lax" />
  <element name="XPath" type="string" />
  </choice>
  <attribute name="Algorithm" use="required" type="anyURI" />
  </complexType>
 <!--
 End Reference
  -->
  <element name="DigestMethod" type="ds:DigestMethodType" />
<complexType name="DigestMethodType" mixed="true">
<sequence>
  <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attribute name="Algorithm" use="required" type="anyURI" />
  </complexType>
  <element name="DigestValue" type="ds:DigestValueType" />
 <!--
 End SignedInfo
  -->
 <!--
 Start KeyInfo
  -->
  <element name="KeyInfo" type="ds:KeyInfoType" />
<complexType name="KeyInfoType" mixed="true">
 <!--
 (1,1) elements from (0,unbounded) namespaces
  -->
<choice maxOccurs="unbounded">
  <element ref="ds:KeyName" />
  <element ref="ds:KeyValue" />
  <element ref="ds:RetrievalMethod" />
  <element ref="ds:X509Data" />
  <element ref="ds:PGPData" />
  <element ref="ds:SPKIData" />
  <element ref="ds:MgmtData" />
  <any namespace="##other" processContents="lax" />
  </choice>
  <attribute name="Id" use="optional" type="ID" />
  </complexType>
  <element name="KeyName" type="string" />
  <element name="MgmtData" type="string" />
  <element name="KeyValue" type="ds:KeyValueType" />
<complexType name="KeyValueType" mixed="true">
<choice>
  <element ref="ds:DSAKeyValue" />
  <element ref="ds:RSAKeyValue" />
  <any namespace="##other" processContents="lax" />
  </choice>
  </complexType>
```

```
  <element name="RetrievalMethod" type="ds:RetrievalMethodType" />
<complexType name="RetrievalMethodType">
<sequence>
  <element ref="ds:Transforms" minOccurs="0" />
  </sequence>
  <attribute name="URI" type="anyURI" />
  <attribute name="Type" use="optional" type="anyURI" />
  </complexType>
 <!--
 Start X509Data
  -->
  <element name="X509Data" type="ds:X509DataType" />
<complexType name="X509DataType">
<sequence maxOccurs="unbounded">
<choice>
  <element name="X509IssuerSerial" type="ds:X509IssuerSerialType" />
  <element name="X509SKI" type="base64Binary" />
  <element name="X509SubjectName" type="string" />
  <element name="X509Certificate" type="base64Binary" />
  <element name="X509CRL" type="base64Binary" />
  <any namespace="##other" processContents="lax" />
  </choice>
  </sequence>
  </complexType>
<complexType name="X509IssuerSerialType">
<sequence>
  <element name="X509IssuerName" type="string" />
  <element name="X509SerialNumber" type="integer" />
  </sequence>
  </complexType>
 <!--
 End X509Data
  -->
 <!--
 Begin PGPData
  -->
  <element name="PGPData" type="ds:PGPDataType" />
<complexType name="PGPDataType">
<choice>
<sequence>
  <element name="PGPKeyID" type="base64Binary" />
  <element name="PGPKeyPacket" type="base64Binary" minOccurs="0" />
  <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
<sequence>
  <element name="PGPKeyPacket" type="base64Binary" />
  <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  </choice>
  </complexType>
 <!--
 End PGPData
  -->
 <!--
 Begin SPKIData
  -->
  <element name="SPKIData" type="ds:SPKIDataType" />
<complexType name="SPKIDataType">
<sequence maxOccurs="unbounded">
  <element name="SPKISexp" type="base64Binary" />
  <any namespace="##other" processContents="lax" minOccurs="0" />
  </sequence>
  </complexType>
 <!--
 End SPKIData
  -->
 <!--
 End KeyInfo
  -->
 <!--
 Start Object (Manifest, SignatureProperty)
```

```
  -->
  <element name="Object" type="ds:ObjectType" />
<complexType name="ObjectType" mixed="true">
 <!--
 add a grep facet
  -->
<sequence minOccurs="0" maxOccurs="unbounded">
  <any namespace="##any" processContents="lax" />
  </sequence>
  <attribute name="Id" use="optional" type="ID" />
  <attribute name="MimeType" use="optional" type="string" />
  <attribute name="Encoding" use="optional" type="anyURI" />
  </complexType>
  <element name="Manifest" type="ds:ManifestType" />
<complexType name="ManifestType">
<sequence>
  <element ref="ds:Reference" maxOccurs="unbounded" />
  </sequence>
  <attribute name="Id" use="optional" type="ID" />
  </complexType>
  <element name="SignatureProperties" type="ds:SignaturePropertiesType" />
<complexType name="SignaturePropertiesType">
<sequence>
  <element ref="ds:SignatureProperty" maxOccurs="unbounded" />
  </sequence>
  <attribute name="Id" use="optional" type="ID" />
  </complexType>
  <element name="SignatureProperty" type="ds:SignaturePropertyType" />
<complexType name="SignaturePropertyType" mixed="true">
 <!--
 (1,1) elements from (1,unbounded) namespaces
  -->
<choice maxOccurs="unbounded">
  <any namespace="##other" processContents="lax" />
  </choice>
  <attribute name="Target" use="required" type="anyURI" />
  <attribute name="Id" use="optional" type="ID" />
  </complexType>
 <!--
 End Object (Manifest, SignatureProperty)
  -->
 <!--
 Start Algorithm Parameters
  -->
 <!--
 Start KeyValue Element-types
  -->
  <element name="DSAKeyValue" type="ds:DSAKeyValueType" />
<complexType name="DSAKeyValueType">
<sequence>
<sequence minOccurs="0">
  <element name="P" type="ds:CryptoBinary" />
  <element name="Q" type="ds:CryptoBinary" />
  </sequence>
  <element name="G" type="ds:CryptoBinary" minOccurs="0" />
  <element name="Y" type="ds:CryptoBinary" />
  <element name="J" type="ds:CryptoBinary" minOccurs="0" />
<sequence minOccurs="0">
  <element name="Seed" type="ds:CryptoBinary" />
  <element name="PgenCounter" type="ds:CryptoBinary" />
  </sequence>
  </sequence>
  </complexType>
  <element name="RSAKeyValue" type="ds:RSAKeyValueType" />
<complexType name="RSAKeyValueType">
<sequence>
  <element name="Modulus" type="ds:CryptoBinary" />
  <element name="Exponent" type="ds:CryptoBinary" />
  </sequence>
  </complexType>
 <!--
```

```
End KeyValue Element-types
 -->
<!--
End Signature
 -->
 </schema>
```

Appendix D (Normative) The ebXML Message Store Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
 <!--
Generated by XML Authority. Conforms to w3c http://www.w3.org/2001/XMLSchema
  -->
<xsd:schema targetNamespace="http://www.oasis-open.org/tc/ebxml-
iic/testing/messageStore" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:mime="http://www.oasis-open.org/tc/ebxml-iic/testing/mime"
xmlns="http://www.oasis-open.org/tc/ebxml-iic/testing/messageStore">
  <xsd:import namespace="http://www.oasis-open.org/tc/ebxml-iic/testing/mime"
schemaLocation="messagestore_mime.xsd" />
<xsd:element name="MessageStore">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element ref="mime:Message" minOccurs="0" maxOccurs="unbounded" />
 </xsd:sequence>
 </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Message Store MIME Message Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
 <!--
Generated by XML Authority. Conforms to w3c http://www.w3.org/2001/XMLSchema
  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.oasis-
open.org/tc/ebxml-iic/testing/mime" xmlns:tns="http://www.oasis-open.org/tc/ebxml-
iic/testing/mime" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://www.oasis-open.org/tc/ebxml-iic/testing/soap">
  <import namespace="http://www.oasis-open.org/tc/ebxml-iic/testing/soap"
schemaLocation="messagestore_soap.xsd" />
<element name="Message">
<complexType mixed="true">
<choice>
  <element ref="tns:MessageContainer" />
  </choice>
  <attribute name="contentType" default="multipart/related" type="string" />
  <attribute name="type" use="optional" type="string" />
  <attribute name="serviceInstanceId" use="required" type="anyURI" />
  <attribute name="id" use="required" type="string" />
  <attribute name="syncType" use="required" type="tns:sync.type" />
  <attribute name="reportingAction" use="required" type="string" />
  <attribute name="serviceName" use="required" type="string" />
  </complexType>
  </element>
<element name="MessageContainer">
<complexType>
<sequence>
  <element ref="soap:Envelope" />
  </sequence>
  <attribute name="contentId" use="optional" type="string" />
  <attribute name="contentType" use="optional" type="string" />
  <attribute name="charset" use="optional" type="string" />
  </complexType>
  </element>
```

```
<element name="PayloadContainer">
<complexType>
  <attribute name="contentId" use="optional" type="string" />
  <attribute name="contentType" use="optional" type="string" />
  <attribute name="charset" use="optional" type="string" />
  </complexType>
  </element>
<simpleType name="sync.type">
<restriction base="NMTOKEN">
  <enumeration value="syncronous" />
  <enumeration value="asyncronous" />
  </restriction>
  </simpleType>
  </schema>
```

Message Store SOAP Message Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
 <!--
Generated by XML Authority. Conforms to w3c http://www.w3.org/2001/XMLSchema
  -->
<xs:schema targetNamespace="http://www.oasis-open.org/tc/ebxml-iic/testing/soap"
xmlns="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://www.oasis-open.org/tc/ebxml-
iic/testing/soap" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:eb="http://www.oasis-open.org/committees/ebxml-msg/schema/msg-header-2 0.xsd">
  <xs:import namespace="http://www.oasis-open.org/committees/ebxml-msg/schema/msg-
header-2_0.xsd" schemaLocation="http://www.oasis-open.org/committees/ebxml-
msg/schema/msg-header-2_0.xsd" />
- <xs:attributeGroup name="encodingStyle">
  <xs:attribute name="encodingStyle" type="tns:encodingStyle" />
  </xs:attributeGroup>
- <!--
 Schema for the SOAP/1.1 envelope

     This schema has been produced using W3C's SOAP Version 1.2 schema
     found at:

     http://www.w3.org/2001/06/soap-envelope

     Copyright 2001 Martin Gudgin, Developmentor.

     Changes made are the following:
     - reverted namespace to http://schemas.xmlsoap.org/soap/envelope/
     - reverted mustUnderstand to only allow 0 and 1 as lexical values

     Original copyright:

     Copyright 2001 W3C (Massachusetts Institute of Technology,
     Institut National de Recherche en Informatique et en Automatique,
     Keio University). All Rights Reserved.
     http://www.w3.org/Consortium/Legal/

     This document is governed by the W3C Software License [1] as
     described in the FAQ [2].

     [1] http://www.w3.org/Consortium/Legal/copyright-software-19980720
     [2] http://www.w3.org/Consortium/Legal/IPR-FAQ-20000620.html#DTD
```

```
  -->
 <!--
 Envelope, header and body
  -->
  <xs:element name="Envelope" type="tns:Envelope" />
<xs:complexType name="Envelope">
<xs:sequence>
  <xs:element ref="tns:Header" />
  <xs:element ref="tns:Body" />
  <xs:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"
/>
  </xs:sequence>
  <xs:anyAttribute namespace="##other" processContents="lax" />
  </xs:complexType>
<xs:group name="optionElements">
<xs:all minOccurs="0">
  <xs:element ref="eb:SyncReply" minOccurs="0" />
  <xs:element ref="eb:MessageOrder" minOccurs="0" />
  <xs:element ref="eb:AckRequested" minOccurs="0" />
  <xs:element ref="eb:Acknowledgment" minOccurs="0" />
  <xs:element ref="eb:ErrorList" minOccurs="0" />
  </xs:all>
  </xs:group>
<xs:element name="Header">
<xs:complexType>
<xs:sequence>
  <xs:element ref="eb:MessageHeader" />
  <xs:group ref="tns:optionElements" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
<xs:complexType name="Header">
<xs:sequence>
  <xs:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"
/>
  </xs:sequence>
  <xs:anyAttribute namespace="##other" processContents="lax" />
  </xs:complexType>
<xs:element name="Body">
<xs:complexType>
<xs:choice>
  <xs:element ref="eb:Manifest" />
  <xs:element ref="eb:StatusRequest" />
  <xs:element ref="eb:StatusResponse" />
  </xs:choice>
  </xs:complexType>
  </xs:element>
<xs:complexType name="Body">
<xs:annotation>
  <xs:documentation>Prose in the spec does not specify that attributes are allowed on
the Body element</xs:documentation>
  </xs:annotation>
<xs:sequence>
  <xs:any namespace="##any" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
  <xs:anyAttribute namespace="##any" processContents="lax" />
  </xs:complexType>
 <!--
 Global Attributes.  The following attributes are intended to be usable via qualified
attribute names on any complex type referencing them.
  -->
<xs:attribute name="mustUnderstand" default="0">
<xs:simpleType>
<xs:restriction base="boolean">
  <xs:pattern value="0|1" />
  </xs:restriction>
  </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="actor" type="anyURI" />
<xs:simpleType name="encodingStyle">
<xs:annotation>
```

```
  <xs:documentation>'encodingStyle' indicates any canonicalization conventions followed
in the contents of the containing element. For example, the value
'http://schemas.xmlsoap.org/soap/encoding/' indicates the pattern described in SOAP
specification</xs:documentation>
  </xs:annotation>
  <xs:list itemType="anyURI" />
  </xs:simpleType>
<xs:complexType name="Fault" final="extension">
<xs:annotation>
  <xs:documentation>Fault reporting structure</xs:documentation>
  </xs:annotation>
<xs:sequence>
  <xs:element name="faultcode" type="QName" />
  <xs:element name="faultstring" type="string" />
  <xs:element name="faultactor" type="anyURI" minOccurs="0" />
  <xs:element name="detail" type="tns:detail" minOccurs="0" />
  </xs:sequence>
  </xs:complexType>
<xs:complexType name="detail">
<xs:sequence>
  <xs:any namespace="##any" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
  <xs:anyAttribute namespace="##any" processContents="lax" />
  </xs:complexType>
  </xs:schema>
```

Appendix E (Normative) The ebXML Test Report Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
 <!--
Generated by XML Authority. Conforms to w3c http://www.w3.org/2001/XMLSchema
  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.oasis-
open.org/tc/ebxml-iic/testReport" xmlns:ebReport="http://www.oasis-open.org/tc/ebxml-
iic/testReport" version="1.0" elementFormDefault="unqualified"
attributeFormDefault="unqualified">
 <!--
 edited with XML Spy v4.3 U (http://www.xmlspy.com) by Michael Kass (NIST)
  -->
<element name="TestReport">
<complexType>
<sequence>
  <element ref="ebReport:MetaData" />
  <element ref="ebReport:TestCase" maxOccurs="unbounded" />
  </sequence>
  </complexType>
  </element>
<element name="MetaData">
<complexType>
<sequence>
  <element ref="ebReport:Description" />
  <element ref="ebReport:Version" />
  <element ref="ebReport:Maintainer" />
  <element ref="ebReport:Location" />
  <element ref="ebReport:PublishDate" />
  <element ref="ebReport:Status" />
  </sequence>
  </complexType>
  </element>
  <element name="Description" type="string" />
  <element name="Version" type="string" />
  <element name="Maintainer" type="string" />
  <element name="Location" type="anyURI" />
  <element name="PublishDate" type="string" />
  <element name="Status" type="string" />
<element name="TestCase">
<complexType>
<sequence>
<sequence maxOccurs="unbounded">
  <element ref="ebReport:TestStep" />
  </sequence>
  </sequence>
  <attribute name="id" use="required" type="string" />
  <attribute name="description" use="optional" type="string" />
  <attribute name="name" use="required" type="string" />
  <attribute name="author" use="optional" type="string" />
  <attribute name="version" use="optional" type="string" />
  <attribute name="requirementReferenceId" use="required" type="anyURI" />
  <attribute name="requirementType" use="required" type="string" />
  <attribute name="testCaseResult" use="required" type="ebReport:testCaseResult.type" />
  <attribute name="specRef" use="required" type="string" />
  </complexType>
  </element>
<element name="TestStep">
<complexType>
<sequence>
<element name="StepResult">
<complexType>
<choice>
```

```
  <element ref="ebReport:Pass" />
  <element ref="ebReport:Fail" />
  </choice>
  </complexType>
  </element>
  </sequence>
  <attribute name="description" use="required" type="string" />
  </complexType>
  </element>
  <element name="ErrorMessage" type="string" />
<simpleType name="mimeHeader.type">
<restriction base="NMTOKEN">
  <enumeration value="MIMEMessageContent-Type" />
  <enumeration value="MIMEMessageStart" />
  <enumeration value="Content-Type" />
  <enumeration value="start" />
  <enumeration value="charset" />
  <enumeration value="type" />
  <enumeration value="wildcard" />
  </restriction>
  </simpleType>
<simpleType name="content.type">
<restriction base="NMTOKEN">
  <enumeration value="XML" />
  <enumeration value="date" />
  <enumeration value="URI" />
  <enumeration value="signature" />
  </restriction>
  </simpleType>
<simpleType name="method.type">
<restriction base="NMTOKEN">
  <enumeration value="xpath" />
  <enumeration value="sha-1" />
  </restriction>
  </simpleType>
<simpleType name="messageContext.type">
<restriction base="NMTOKEN">
  <enumeration value="true" />
  <enumeration value="false" />
  </restriction>
  </simpleType>
<simpleType name="requirement.type">
<restriction base="NMTOKEN">
  <enumeration value="required" />
  <enumeration value="stronglyrecommended" />
  <enumeration value="recommended" />
  <enumeration value="optional" />
  </restriction>
  </simpleType>
<simpleType name="testCaseResult.type">
<restriction base="NMTOKEN">
  <enumeration value="pass" />
  <enumeration value="fail" />
  <enumeration value="untested" />
  </restriction>
  </simpleType>
<simpleType name="failure.type">
<restriction base="NMTOKEN">
  <enumeration value="preConditionTest" />
  <enumeration value="assertionTest" />
  <enumeration value="undetermined" />
  </restriction>
  </simpleType>
<element name="Container">
<complexType>
<simpleContent>
<extension base="string">
  <attribute name="content-id" use="optional" type="string" />
  <attribute name="content-location" use="optional" type="string" />
  </extension>
  </simpleContent>
```

```
  </complexType>
  </element>
<element name="Message">
<complexType>
<sequence>
  <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attribute name="id" use="required" type="ID" />
  </complexType>
  </element>
<element name="Pass">
  <complexType />
  </element>
<element name="Fail">
<complexType>
  <attribute name="failType" use="optional" type="ebReport:failure.type" />
  </complexType>
  </element>
<element name="SoapFault">
<complexType>
<sequence>
  <element name="faultcode" type="QName" />
  <element name="faultstring" type="string" />
  <element name="faultactor" type="anyURI" minOccurs="0" />
  <element name="detail" type="NCName" minOccurs="0" />
  </sequence>
  </complexType>
  </element>
<complexType name="detail">
<sequence>
  <any namespace="##any" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <anyAttribute namespace="##any" processContents="lax" />
  </complexType>
  <element name="ebXMLError" type="string" />
  </schema>
```

Appendix F (Normative) Service Related Message Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
 <!--
Generated by XML Authority. Conforms to w3c http://www.w3.org/2001/XMLSchema
  -->
<xsd:schema xmlns:eb="http://www.oasis-open.org/committees/ebxml-msg/schema/msg-header-
2 0.xsd" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:import namespace="http://www.oasis-open.org/committees/ebxml-msg/schema/msg-
header-2 0.xsd" schemaLocation="http://www.oasis-open.org/committees/ebxml-
msg/schema/msg-header-2_0.xsd" />
  <xsd:import namespace="http://schemas.xmlsoap.org/soap/envelope/"
schemaLocation="http://schemas.xmlsoap.org/soap/envelope/" />
<xsd:element name="InitiatorRequest">
<xsd:complexType>
<xsd:sequence>
  <xsd:element name="CPAId" />
  <xsd:element name="ConversationId" />
  <xsd:element name="MessageId" />
  </xsd:sequence>
  </xsd:complexType>
  </xsd:element>
<xsd:element name="PayloadVerifyResponse">
<xsd:complexType>
<xsd:sequence>
  <xsd:element name="Result" type="xsd:boolean" />
  </xsd:sequence>
  </xsd:complexType>
  </xsd:element>
<xsd:element name="ErrorAppNotifyResponse">
<xsd:complexType>
<xsd:sequence>
  <xsd:element ref="eb:ErrorList" maxOccurs="unbounded" />
  </xsd:sequence>
  </xsd:complexType>
  </xsd:element>
<xsd:element name="ErrorURLNotifyResponse">
<xsd:complexType>
<xsd:sequence>
  <xsd:element ref="eb:ErrorList" maxOccurs="unbounded" />
  </xsd:sequence>
  </xsd:complexType>
  </xsd:element>
<xsd:element name="ConfiguratorRequest">
<xsd:complexType>
<xsd:sequence>
<xsd:choice minOccurs="0">
  <xsd:element ref="CPAId" />
  <xsd:element ref="CPA" />
  </xsd:choice>
  <xsd:element ref="Mode" minOccurs="0" />
  <xsd:element name="ResponseURL" minOccurs="0" />
  </xsd:sequence>
  </xsd:complexType>
  </xsd:element>
<xsd:element name="ConfiguratorResponse">
<xsd:complexType>
<xsd:sequence>
  <xsd:element ref="Status" minOccurs="0" />
  </xsd:sequence>
  </xsd:complexType>
  </xsd:element>
```

```
<xsd:element name="CPA">
<xsd:complexType>
<xsd:sequence>
  <xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"
/>
  </xsd:sequence>
  </xsd:complexType>
  </xsd:element>
  <xsd:element name="Status" type="xsd:boolean" />
  <xsd:element name="CPAId" type="non-empty-string" />
  <xsd:element name="Mode" type="non-empty-string" />
  <xsd:element name="ConversationId" type="non-empty-string" />
  <xsd:element name="MessageId" type="non-empty-string" />
<xsd:simpleType name="non-empty-string">
<xsd:restriction base="xsd:string">
  <xsd:minLength value="1" />
  </xsd:restriction>
  </xsd:simpleType>
<xsd:element name="TestMessage">
<xsd:complexType>
<xsd:choice>
  <xsd:element ref="InitiatorRequest" />
  <xsd:element ref="PayloadVerifyResponse" />
  <xsd:element ref="ConfiguratorRequest" />
  <xsd:element ref="ConfiguratorResponse" />
  </xsd:choice>
  </xsd:complexType>
  </xsd:element>
  </xsd:schema>
```

Appendix G Terminology

Several terms used in this specification are borrowed from the Conformance Glossary (OASIS, [ConfGlossary]) and also from the Standards and Conformance Testing Group at NIST. [ConfCertModelNIST]. They are not reported in this glossary, which only reflects (1) terms that are believed to be specific to – and introduced by - the ebXML Test Framework, or (2) terms that have a well understood meaning in testing literature (see above references) and may have additional properties in the context of the Test Framework that is worth mentioning.

Term	Definition
Asymmetric testing	Interoperability testing where all parties are not equally tested for the same features. An asymmetric interoperability test suite is typically driven from one party, and will need to be executed from every other party in order to evenly test for all interoperability features between candidate parties.
Base CPA	Required by both the conformance and interoperabililty test suites that describe both the Test Driver and Test Service Collaboration Protocol Profile Agreement. This is the "bootstrap" configuration for all messaging between the testing and candidate ebXML applications. Each test suite will define additional CPAs. How the base CPA is represented to applications is implementation specific.
Candidate Implementation	(or Implementation Under test): The implementation (realization of a specification) used as a target of the testing (e.g. conformance testing).
Conformance	Fulfillment of an implementation of all requirements specified; adherence of an implementation to the requirements of one or more specific standards or specifications.
Connection mode (Test Driver in)	In connection mode and depending on the test harness, the test driver will interact with other components by directly generating ebXML messages at transport level (e.g. generates HTTP envelopes).
Interoperability profile	A set of test requirements for interoperability which is a subset of all possible interoperability requirements, and which usually exercises features that correspond to specific user needs.
Interoperability Testing	Process of verifying that two implementations of the same specification, or that an implementation and its operational environment, can interoperate according to the requirements of an assumed agreement or contract. This contract does not belong necessarily to the specification, but its terms and elements should be defined in it with enough detail, so that such a contract, combined with the specification, will be sufficient to determine precisely the expected behavior of an implementation, and to test it.
Local Reporting mode (Test Service in)	In this mode (a sub-mode of Reporting), the Test Service is installed on the same host as the Test Driver it reports to, and executes in the same process space. The notification uses the *Receive* interface of the Test Driver, which must be operating in service mode.

Loop mode (Test Service in)	When a test service is in loop mode, it does not generate notifications to the test driver. The test service only communicates with external parties via the message handler.
MSH	Message Service Handler, an implementation of ebXML Messaging Services
Reporting mode (Test Service in)	A test service is deployed in reporting mode, when it notifies the test driver of invoked actions. This notification usually contains material from received messages.
Profile	A profile is used as a method for defining subsets of a specification by identifying the functionality, parameters, options, and/or implementation requirements necessary to satisfy the requirements of a particular community of users. Specifications that explicitly recognize profiles should provide rules for profile creation, maintenance, registration, and applicability.
Remote Reporting mode (Test Service in)	In this mode (a sub-mode of Reporting), the Test Service is deployed on a different host than the Test Driver it reports to. The notification is done via messages to the Test Driver, which is operating in connection mode.
Service mode (Test Driver in)	The Test Driver invokes actions in the test service via a programmatic interface (as opposed to via messages). The Test Service must be in local reporting mode.
Specification coverage	Specifies the degree that the specification requirements are satisfied by the set of test requirements included in the test suite document. Coverage can be full, partial or none.
Test actions	(Or Test Service actions). Standard functions available in the test service to support most test cases.
Test case	In the TestFramework, a test case is a sequence of discrete test steps, aimed at verifying a test requirement.
Test Requirements coverage	Specifies the degree that the test requirements are satisfied by the set of test cases listed in the test suite document. Coverage can be full, contingent, partial or none.

4912

Appendix H References

H.1 Normative References

[ConfCertModelNIST] Conformance Testing and Certification Model for Software Specifications. L. Carnahan, L. Rosenthal, M. Skall. ISACC '98 Conference. March 1998

[ConfCertTestFrmk] Conformance Testing and Certification Framework. L. Rosenthal, M. Skall, L. Carnahan. April 2001

[ConfReqOASIS] Conformance Requirements for Specifications. OASIS Conformance Technical Committee. March 2002.

[ConfGlossary] Conformance Glossary. OASIS Conformance TC, L. Rosenthal. September 2000.

[RFC2119] Key Words for use in RFCs to Indicate Requirement Levels, Internet Engineering Task Force, March 1997

[RFC2045] Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies, N Freed & N Borenstein, Published November 1996

[RFC2046] Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. N. Freed, N. Borenstein. November 1996.

[RFC2387] The MIME Multipart/Related Content-type. E. Levinson. August 1998.

[RFC2392] Content-ID and Message-ID Uniform Resource Locators. E. Levinson, August 1998

[RFC2396] Uniform Resource Identifiers (URI): Generic Syntax. T Berners-Lee, August 1998

[RFC2821] Simple Mail Transfer Protocol, J. Klensin, Editor, April 2001 Obsoletes RFC 821

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol, HTTP/1.1", June 1999.

[SOAP] W3C-Draft-Simple Object Access Protocol (SOAP) v1.1, Don Box, DevelopMentor; David Ehnebuske, IBM; Gopal Kakivaya, Andrew Layman, Henrik Frystyk Nielsen, Satish Thatte, Microsoft; Noah Mendelsohn, Lotus Development Corp.; Dave Winer, UserLand Software, Inc.; W3C Note 08 May 2000, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[SOAPAttach] SOAP Messages with Attachments, John J. Barton, Hewlett Packard Labs; Satish Thatte and Henrik Frystyk Nielsen, Microsoft, Published Oct 09 2000 http://www.w3.org/TR/2000/NOTE-SOAP-attachments-20001211

[XLINK] W3C XML Linking Recommendation, http://www.w3.org/TR/2001/REC-xlink-20010627/

[XML] W3C Recommendation: Extensible Markup Language (XML) 1.0 (Second Edition), October 2000, http://www.w3.org/TR/2000/REC-xml-20001006

[XMLC14N] W3C Recommendation Canonical XML 1.0, http://www.w3.org/TR/2001/REC-xml-c14n-20010315

[XMLNS] W3C Recommendation for Namespaces in XML, World Wide Web Consortium, 14 January 1999, http://www.w3.org/TR/1999/REC-xml-names-19990114/

[XMLDSIG] Joint W3C/IETF XML-Signature Syntax and Processing specification, http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/.

[XPointer] XML Pointer Language (XPointer) Version 1.0, W3C Candidate Recommendation 11 September 2001, http://www.w3.org/TR/2001/CR-xptr-20010911/

H.2 Non-Normative References

[ebTestFramework] ebXML Test Framework specification, Version 1.0, Technical Committee Specification, March 4, 2003, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ebxml-iic

[ebMS] ebXML Messaging Service Specification, Version 2.0, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ebxml-msg

[ebMSInteropTests] ebXML MS V2.0 Basic Interoperability Profile Test Cases, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ebxml-iic

[ebMSConfTestSuite] ebXML MS V2.0 Conformance Test Suite, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ebxml-iic

[ebMSInteropReqs] ebXML MS V2.0 Interoperability Test Requirements, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ebxml-iic

[XMLSchema] W3C XML Schema Recommendation,
http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/
http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/
http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

[ebCPP] ebXML Collaboration Protocol Profile and Agreement specification, Version 1.0, published 10 May, 2001, http://www.ebxml.org/specs/ebCCP.doc

[ebBPSS] ebXML Business Process Specification Schema, version 1.0, published 27 April 2001, http://www.ebxml.org/specs/ebBPSS.pdf.

[ebRS] ebXML Registry Services Specification, version 2.0, published 6 December 2001 http://www.oasis-open.org/committees/regrep/documents/2.0/specs/ebrs.pdf, published, 5 December 2001.

Appendix I Acknowledgments

The authors wish to acknowledge the support of the members of the OASIS ebXML IIC TC who contributed ideas, comments and text to this specification by the group's discussion eMail list, on conference calls and during face-to-face meetings.

I.1 IIC Committee Members

Jacques Durand, Fujitsu <jdurand@fsw.fujitsu.com>
Jeffery Eck, Global Exchange Services <Jeffery.Eck@gxs.ge.com>
Hatem El Sebaaly, IPNet Solutions <hatem@ipnetsolutions.com>
Aaron Gomez, Drummond Group Inc. <aaron@drummondgroup.com>
Michael Kass, NIST <michael.kass@nist.gov>
Matthew MacKenzie, Individual <matt@mac-kenzie.net>
Monica Martin, Sun Microsystems <monica.martin@sun.com>
Tim Sakach, Drake Certivo <tsakach@certivo.net>
Jeff Turpin, Cyclone Commerce <jturpin@cyclonecommerce.com>
Eric van Lydegraf, Kinzan <ericv@kinzan.com>
Pete Wenzel, SeeBeyond <pete@seebeyond.com>
Steven Yung, Sun Microsystems <steven.yung@sun.com>
Boonserm Kulvatunyou, NIST <serm@nist.gov>

Appendix J Revision History

Rev	Date	By Whom	What
cs-10	2003-03-07	Michael Kass	Initial version

Appendix K Notices