This document describes the scripting language for event-based log analysis, testing and monitoring of processes.
Technical Committee members should send comments on this specification to the Technical Committee’s email list. Others should send comments to the Technical Committee by using the “Send A Comment” button on the Technical Committee’s web page at http://www.oasis-open.org/committees/tamie/.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (http://www.oasis-open.org/committees/tamie/ipr.php).

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

Copyright © OASIS® 2008-2010. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

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

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS’ procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name "OASIS" is a trademark of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see http://www.oasis-open.org/who/trademark.php for above guidance.
# Table of Contents

1 Introduction ........................................................................................................................................... 7
   1.1 Intended Audience .......................................................................................................................... 7
   1.2 Conventions ....................................................................................................................................... 7

2 Objectives .............................................................................................................................................. 9
   2.1 Testing and Monitoring of Business Processes and Transactions .................................................. 9
   2.2 Event-Driven Execution .................................................................................................................... 9
   2.3 Leverage of existing XML script languages .................................................................................... 9
   2.4 Live and Deferred Testing ............................................................................................................... 9
   2.5 Outputs: Effect vs. Events .............................................................................................................. 10

3 Language Constructs and Semantics ................................................................................................. 11
   3.1 Scriplets ........................................................................................................................................... 11
      3.1.1 Scriplet ........................................................................................................................................ 11
      3.1.2 param ........................................................................................................................................ 11
      3.1.3 start .......................................................................................................................................... 12
   3.2 The Script Package and its Execution Context ................................................................................. 12
      3.2.1 script-package ............................................................................................................................ 12
      3.2.2 execution-context ....................................................................................................................... 12
      3.2.3 start-with ................................................................................................................................... 13
      3.2.4 nms-binding ............................................................................................................................... 13
      3.2.5 event-board ............................................................................................................................... 13
      3.2.6 filter .......................................................................................................................................... 15
      3.2.7 condition ................................................................................................................................... 15
      3.2.8 execution-context/var ............................................................................................................... 15
   3.3 Event Catching .................................................................................................................................. 15
      3.3.1 catch ......................................................................................................................................... 16
      3.3.2 catch/match ............................................................................................................................... 17
      3.3.3 catch/date-time ........................................................................................................................... 19
      3.3.4 mask ......................................................................................................................................... 19
   3.4 Conditional Statements .................................................................................................................... 20
      3.4.1 if ............................................................................................................................................... 20
      3.4.2 decide ....................................................................................................................................... 20
      3.4.3 decide/if ..................................................................................................................................... 21
      3.4.4 decide/else-if ............................................................................................................................. 21
      3.4.5 decide/else ............................................................................................................................... 21
   3.5 Variables .......................................................................................................................................... 22
      3.5.1 var ............................................................................................................................................ 22
      3.5.2 execution-context/var ............................................................................................................. 24
   3.6 The “Effect” of Various Constructs .................................................................................................. 25
      3.6.1 In-line effects ............................................................................................................................ 25
      3.6.2 Inserting the value of expressions in effects: eval ................................................................. 25
      3.6.3 Generating effect with CDATA ............................................................................................ 25
      3.6.4 Concurrency Semantics .......................................................................................................... 25
   3.7 Iterations ......................................................................................................................................... 26
      3.7.1 loop ......................................................................................................................................... 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7.2 loop/on-start</td>
<td>29</td>
</tr>
<tr>
<td>3.7.3 loop/on-start/loop-var</td>
<td>30</td>
</tr>
<tr>
<td>3.7.4 loop/next-val</td>
<td>30</td>
</tr>
<tr>
<td>3.7.5 loop/on-final</td>
<td>30</td>
</tr>
<tr>
<td>3.8 Message Handling</td>
<td>30</td>
</tr>
<tr>
<td>3.8.1 message</td>
<td>30</td>
</tr>
<tr>
<td>3.8.2 post</td>
<td>31</td>
</tr>
<tr>
<td>3.9 Exiting</td>
<td>31</td>
</tr>
<tr>
<td>3.9.1 exit</td>
<td>31</td>
</tr>
<tr>
<td>3.9.2 Exiting from Variable definitions</td>
<td>32</td>
</tr>
<tr>
<td>3.9.3 Exiting from Conditional Statements</td>
<td>33</td>
</tr>
<tr>
<td>3.10 Time Control</td>
<td>33</td>
</tr>
<tr>
<td>3.10.1 Updating the VP-time</td>
<td>33</td>
</tr>
<tr>
<td>3.10.2 sleep</td>
<td>34</td>
</tr>
<tr>
<td>3.10.3 Synchronous vs. Asynchronous Invocations</td>
<td>34</td>
</tr>
<tr>
<td>3.11 Adapters</td>
<td>34</td>
</tr>
<tr>
<td>3.11.1 call-adapter</td>
<td>34</td>
</tr>
<tr>
<td>3.11.2 call-adapter/with-param</td>
<td>35</td>
</tr>
<tr>
<td>3.12 Functions</td>
<td>35</td>
</tr>
<tr>
<td>3.12.1 decide</td>
<td>36</td>
</tr>
<tr>
<td>3.12.2 decide/if</td>
<td>36</td>
</tr>
<tr>
<td>3.12.3 decide/else-if</td>
<td>36</td>
</tr>
<tr>
<td>3.12.4 decide/else</td>
<td>36</td>
</tr>
<tr>
<td>3.12.5 exit</td>
<td>36</td>
</tr>
<tr>
<td>3.12.6 if</td>
<td>36</td>
</tr>
<tr>
<td>3.12.7 var</td>
<td>37</td>
</tr>
<tr>
<td>4 The Base Language Subset</td>
<td>39</td>
</tr>
<tr>
<td>4.1 The Script-package</td>
<td>39</td>
</tr>
<tr>
<td>4.1.1 script-package</td>
<td>39</td>
</tr>
<tr>
<td>4.1.2 execution-context</td>
<td>39</td>
</tr>
<tr>
<td>4.1.3 nms-binding</td>
<td>39</td>
</tr>
<tr>
<td>4.1.4 start-with</td>
<td>39</td>
</tr>
<tr>
<td>4.1.5 condition</td>
<td>39</td>
</tr>
<tr>
<td>4.1.6 execution-context/var</td>
<td>40</td>
</tr>
<tr>
<td>4.1.7 Scriplets</td>
<td>40</td>
</tr>
<tr>
<td>4.1.8 scriptlet</td>
<td>40</td>
</tr>
<tr>
<td>4.1.9 param</td>
<td>40</td>
</tr>
<tr>
<td>4.1.10 catch</td>
<td>40</td>
</tr>
<tr>
<td>4.1.11 catch/match</td>
<td>41</td>
</tr>
<tr>
<td>4.1.12 condition</td>
<td>41</td>
</tr>
<tr>
<td>4.1.13 decide</td>
<td>41</td>
</tr>
<tr>
<td>4.1.14 decide/if</td>
<td>41</td>
</tr>
<tr>
<td>4.1.15 decide/else-if</td>
<td>41</td>
</tr>
<tr>
<td>4.1.16 decide/else</td>
<td>42</td>
</tr>
<tr>
<td>4.1.17 eval</td>
<td>42</td>
</tr>
<tr>
<td>4.1.18 exit</td>
<td>42</td>
</tr>
</tbody>
</table>
1 Introduction

This document is a specification of the eTSM script language..

Two requirements motivate a standard representation of test cases:

- Only a formal and processable representation of test suites, along with the operational model behind it, can provide users with a precise unambiguous operational semantics of test cases and of the actual verification they perform. The way a test case will be interpreted e.g. by a test driver, must be clear to all parties involved in a testing round. Standardizing this representation lays ground for a common understanding by a broad community of users.

- it is desirable that testing be automated, so that test case execution can be done at low cost, and repeated if required. Test case definitions should also be portable from one test engine to the other. This requires a processable representation, at higher level than programming languages, as well as independent from other mark-ups used for controlling some of the layers of e-Business (Web services choreography, business transactions...) as implementations of these may be themselves the target for testing.

Organization of the document:

Section 2 describes ...

Section 3 describes ...

Section 4 explains ...

Section 5 conformance section; t....

Appendices provide a glossary of important terms, references, a listing of related reading material and a worked example.

1.1 Intended Audience

The primary audience for this document is the authors of specifications and the writers of test suites and monitoring scripts for processes and event logs.

1.2 Conventions

The following conventions are used to represent the XML mark-up:

<example
   id = 'xsd:normalizedString'
   text ? = 'xsd:string'>
   Content: child ?, sibling *, 'xsd:Any' *
</example>

In the example representation above there is an element called “example” (not a real element, just an example to illustrate the XML representation convention used in this specification). The element has a mandatory attribute, shown without any following symbolic characters to signify that it is mandatory, called “id” whose content type is the W3C XML Schema datatype “normalizedString”, shown as
“normalizedString” prefixed with the letters “xsd:” to denote a W3C XML Schema datatype. The element called “example” has another attribute named “text” which is shown to be optional by the symbol “?” after the name: “text ?”. The “example” element has element children called “child” and “sibling” listed after the label “Content:” in italics. The “*” asterisk character signifies that the child element “sibling” has multiple cardinality and is optional. The “?” question mark character after the child element “child” signifies that this element is optional. If the “example” element did not have elements as children but if its content had been of datatype “integer” then the content would have been represented as “Content: xsd:integer”.

Finally, the example contains in the Content an 'xsd:any' which indicates that the element concerned can be extended with zero or more additional elements as long as they do not belong to the same namespace as the present markup. These extra elements will not be validated by W3C XML Schema validation, they are skipped.

XML Representation:

<another-example
    choice ? = 1|2
    {any attributes with non-schema namespace . . .}>
    Content: child +, next, (sibling|other) ?
</another-example>

In the next example, above, an element called “another-example” has an optional attribute named “choice” whose content can be either the literal value “1” or the literal value “2”. The child element named “child” is both mandatory and multiple cardinality, symbolized by the plus character “+” after the name. Another child element “next” follows the element called “child” and is mandatory and singular cardinality, denoted by the absence of any character following the name. There is also a choice of one of two possible other child elements named “sibling” and “other”, the choice being denoted by a single vertical line character between the two elements and the group of choices being shown with round brackets surrounding these two choices. Following the brackets there is a “?” question-mark character to denote that either of these is optional. The commas separating the child element names would denote that there is no required sequence.

.
2 Objectives

eTSM is addressing the testing needs of distributed applications and interactive business processes, the behavior of which can be traced and monitored using the notion of event. A deliberate design choice in eTSM is to leverage the growing support for XML in terms of tooling and processing, both as test material (e.g. event logs, configuration documents, metadata definitions) and as reporting format (e.g. structured test reports subject to subsequent processing e.g. HTML rendering, embedding of input test material).

The objectives can be categorized as follows:

2.1 Testing and Monitoring of Business Processes and Transactions

- validating process / bus Tx run-time against their definitions
- generate alerts and metrics for run-time SLA monitoring of processes / transactions (e.g. BAM)
- analytics of logs of processes and business transactions for reports and BI.

2.2 Event-Driven Execution

The eTSM script language is designed for testing and monitoring processes or business transactions of various kinds, and more particularly for analyzing and validating event patterns that are generated by these processes. To this extent, eTSM may be described as an event-processing language. It assumes a particular XML format - or wrapper - for these events.

eTSM is an XML markup language, and has extensive provisions for leveraging existing expression languages. One such language is XPath, which is the default dialect used in various expressions and conditions of eTSM, although it is open to other dialects.

2.3 Leverage of existing XML script languages

eTSM is designed so that it leverages existing XML script languages for special features such as logical conditions and event selection. The default language for all logical expressions over XML content is XPath, along with its existing function libraries (e.g. advanced set of functions for time and duration management). eTSM is therefore designed as a "platform" language that can be extended with other script languages covering specialized features. These extending scripts are called "dialects".

eTSM is however open to other XML dialects than XPath1.0 or XPath2.0, e.g. could use XQuery instead or in combination. Another XML dialect is XSLT, which will be the default dialect for writing external functions.

A base subset of eTSM will be defined so that it is implementable entirely in XSLT.

2.4 Live and Deferred Testing

The language is designed so that the same scripts can be used either in live monitoring mode, or in analysis of past events (deferred mode) or yet in mixed situation e.g. starting live, until the test engine has to be stopped for some reason, then resuming over the log of events registered during shut-down, and catching up again in live mode.

From the viewpoint of script execution semantics, "live" and "deferred" are not distinguished: the same script must be executable on either live or "stale" input ("stale" input meaning the events are already logged - this is also called "deferred execution"), depending on how its virtual present date/time is set (i.e.
virtual execution date/time). This ensures flexibility to the execution model in order to handle various monitoring contexts and situations, some mixing both execution modes:

- a script may start executing "deferred" with its events already partially logged, and then catch-up with the logging of events and continue "live".

- conversely, a script may start live, and if its execution engine is interrupted for some reason, may resume in deferred mode its analysis of events that have already been logged while the engine was stopped. Then it may catch-up with events and eventually go live again.

This flexibility is obtained thanks to a notion of "virtual present time (VP-time)", which can be different from the actual execution time. When a script starts executing, it is given a "start date/time" which represents the VP-time, which could be any date in the past and by default is same as the actual present. The VP-time - also called "VP cursor" - slides from this start date/time and may "catch-up" with the actual present.

### 2.5 Outputs: Effect vs. Events

Two kinds of output are to be produced by scripts:

- XML documents. Such a document may be a well-formed test report ready for further rendering or processing.

- Events (or messages), posted to an Event Board. A log of such events may be the preferred format for monitoring or testing results. Outputs in form of events also allow for dynamic cooperation between monitoring scripts, as well as pipe-line processing of test material.
3 Language Constructs and Semantics

3.1 Scriplets

Scriplets are the basic containers of eTSM statements. The chaining and branching of scriplets define monitoring or testing workflows that may execute concurrently. Their execution can mix event-driven mode and conventional algorithmic controls.

3.1.1 Scriplet

Role: A scriplet is the basic unit of script. A scriplet has the following semantics:

(a) scoping unit for variables (and generally scoping unit for variable assignment as well) (isn't that the same??).

(b) invocation semantics: a scriplet is a named unit of script that can be invoked from other scriplets, with parameter passing.

(c) thread semantics: a scriplet executes its operations in one thread. However, a scriplet can invoke another scriplet "concurrently", meaning that both scriplets execute concurrently afterward. There is an implicit "joining" of all scriplets started from a (parent) scriplet: the parent only completes its execution after all children scriplets - either serial or concurrent - terminate.

XML Representation:

```xml
<scriplet
  name   = 'xsd:normalizedString'>
  Content: param *, (call-adapter | catch | decide | eval | exit | if |
            loop | message | post | sleep | start | var | 'xsd:Any' ) *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</scriplet>
```

3.1.2 param

Role: defines a parameter for a scriplet.

XML Representation:

```xml
<param
  name   = 'xsd:normalizedString'
  type ? = 'xsd:normalizedString'/>
```

A parameter is a variable with the additional property that its value can be set by the caller when the scriplet is invoked. Like a variable, a parameter can be shadowed by subsequent declarations inside constructs contained by the scriplet.
3.1.3 start

Role: start a scriplet execution.

XML Representation:

```xml
<start
    scriplet   = 'xsd:normalizedString'
    async ? = 'xsd:boolean'
    datetime ? = 'xsd:DateTime'
    bubble-exit ? = 'xsd:boolean'>

    Content: with-param *

</start>
```

*scriplet*: the name of the invoked scriplet (which must belong to the same scriplt package as the invoking scriplet)

*async*:
- when async="false", the started scriplet is invoked synchronously. This means that the next statement (after `<start>`) in the invoking scriplet, will execute at a VP-time that is same as the VP-time set at the end of the invoked scriplet.
- when async="true", the started scriplet is invoked asynchronously. This means that the next statement in the invoking scriplet, will execute at a VP-time that is same as the VP-time of the `<start>` statement.

*bubble-exit*: if "true", any exiting from the invoked scriplet, will bubble-up in the invoking scriplet, causing its termination when the `<start>` statement completes. If "false", the invoking scriplet will resume its execution after the `<start>` statement completes.

3.2 The Script Package and its Execution Context

The highest unit of execution in eTSM is the Script package. A script package contains an execution context element, zero or more scriplets, and zero or more functions.

3.2.1 script-package

Role: organizes scriplets as a package intended as a unit of deployment and defining a referential boundaries for scriplets.

XML Representation:

```xml
<script-package
    name ? = 'xsd:normalizedString'
    schemaVersionId ? = 'xsd:normalizedString'>

    Content: execution-context ?, (scriplet | function ) *

</script-package>
```

3.2.2 execution-context

Role: The executionContext is associated with a script package. It specifies how the scriplets inside the package bind to a specific execution (or monitoring) environment. It specifies bindings of different kinds:
namespaces, event boards. It specifies which scriplet should be executed initially. That is also where some global variable value could be specified. By expressing context-dependent details outside the scriplets, the latter can be reused more easily from one package to the other.

XML Representation:
<execution-context>
   Content: nms-binding *, start-with , var *, event-board *
</execution-context>

### 3.2.3 start-with

**Role:** Indicates which scriplet among those at top level in the package must be started first

XML Representation:
<start-with
datetime ? = 'xsd:DateTime'
   scriplet   = 'xsd:normalizedString'/>

### 3.2.4 nms-binding

**Role:** Container for namespace bindings. A namespace binding defines a prefix associated with a namespace, that is usable inside XML fragments and other expressions (e.g. XPath expressions) used inside scriplets and functions

XML Representation:
<nms-binding
   prefix   = 'xsd:normalizedString'>
   Content: 'xsd:anyURI'
</nms-binding>

### 3.2.5 event-board

**NOTE:** the concept of Event Board is more fully described in the Event Management section, along with the overall Event model. The present section is only concerned with the syntactic aspect of defining an event board in the context of a script package definition.

**Role:** defines a "logical" event board for a script package, that will bind to a physical event store (e.g. an event queue). Such an event board is "declared" in the execution-context of the script package. Parts of this declaration is processed by the scriplet engine, but the binding to an external event source (or store) is performed by an event manager function beyond the scope of scriplet processing. Event filters defined in the declaration provide a "view" over the bound event source, of interest for this script package. This view defines the logical event board.

XML Representation:
<event-board
   name ? = 'xsd:normalizedString'
   event-store ? = 'xsd:normalizedString'
   mode ? = ('source'|'sink'|'source-sink')>
   Content: filter *
</event-board>

**name:** the name used for this event board inside this script package.
event-store: a reference to an actual event source to which this event board is to be bound.
mode: defines the mode of use for this event board: as a source of events (or "read-only"), as a sink (or "write-only") or as both.

Besides these attributes, the event-board element contains one or more filters. In a source or source-sink mode of use, the filter acts as a selector on the physical event store.

In the example 1 below, the declared "PO" event board is selecting all events related to "Purchase orders", i.e. events of types: PurchaseOrder, POrderResponse, POrderCancel, POrderEdit. The event type is here indicated by the presence of an element with related name inside the SOAP Body element.

Example 1:

```
<!-- binding an event board name to an actual event board -->
<event-board name="PO" event-store="(URI of some physical store)"
mode="source">

<!-- any event from the physical source that matches anyone of the following
filters is posted to this eBoard -->

<filter name="POrder">
  <!-- this filter selects POs. -->
  <condition>//content//s:Envelope/s:Body/app:PurchaseOrder</condition>
</filter>

<filter name="POrderResponse">
  <!-- this filter selects PO responses. -->
  <condition>//content//s:Envelope/s:Body/app:POrderResponse</condition>
</filter>

<filter name="POrderCancel">
  <condition>//content//s:Envelope/s:Body/app:POrderCancel</condition>
</filter>

</event-board>
```

An event board declaration my index [some of its] events using a "key". Such indexing is indicated by the presence of an event-board/filter/condition/@key attribute. In the Example 2 below, a key expression is defined for all selected events, the value of which will be defined by a different expression depending on the event type. Such indexing will enhance the performance (or scalability) of subsequent CATCH operations on this event board.

Example 2:

```
<!-- binding an event board name to an actual event board -->
<event-board name="PO" event-store="(URI of some physical store)"
mode="source">

<!-- any event that matches anyone of the following filters is posted to this
eBoard. The key value is extracted from each event selected by this filter,
meaning that only events with the same key value will be assigned to the same
partition entry. Later, a CATCH operator will be able to mention a key value
to reduce the scope of its catching. -->

<filter name="POrder">
  <!-- this filter selects POs. -->
  <condition
    key="//content//s:Envelope/s:Body/app:POrder/@ref"//content//s:Envelope/s:Bod
y/app:PurchaseOrder</condition>
</filter>
```
3.2.6 filter

**Role:** defines an event filter.

**XML Representation:**

```xml
<filter name="POrderResponse">
  <!-- this filter selects PO responses. -->
  <condition
    key="//content//s:Envelope/s:Body/app:POrderResponse/@ref"/>
</filter>

<filter name="POrderCancel">
  <condition
    key="//content//s:Envelope/s:Body/app:POrderCancel/@ref"/>
</filter>
</event-board>
```

A filter will select a particular class of events and add them to the event board.

3.2.7 condition

**Role:** A condition is selecting events on behalf of a filter.

**XML Representation:**

```xml
<condition
  key="xsd:normalizedString">
  Content: '"xsd:normalizedString'
</condition>
```

A condition defines an expression (e.g. XPath) used to select events that are qualified for posting on the event board. A condition may define a key (@key) that will be associated with the events it selects. The attribute @key defines an expression which, when evaluated over each selected event, returns the key value associated with this event.

3.2.8 execution-context/var

**Role:** Defines a variable global to the script package.

[see Section 3.6]

3.3 Event Catching

All event inputs to eTSM scripts are handled by a single eTSM operator: Catch.

The same catch operator is used in two different contexts of event catching:
• **Live execution.** A live execution operates over "live" event input, i.e. events may not have occurred yet. In that case, "catch" is a synchronization operator that will cause a script execution to wait for the expected event(s).

• **Deferred execution.** A deferred execution operates over "stale" input, i.e. events that have already occurred and are already logged in the event board. In that case, "catch" acts as a query to the event board.

Only events that are timestamped before the VP-time (virtual present time) are visible to a catch operator. However, the catch operator has the effect to "slide" the VP cursor forward in time, as long as the VP-cursor remains anterior to the actual present time (time of execution).

The catch operator allows for catching a combination of correlated events, where each one of these may require a different selection condition. Such a combination is called an "event pattern".

For example, the following catch statement will select a combination of correlated messages: a Purchase Order, followed by a POResponse, followed by a POReceipt. These three message items represent an event pattern. The Catch operation will try for a maximum of 3 days and 12h to get such a pattern, starting from the current date/time (VP-time):

```xml
<catch eb="POs" tryfor="P3DT12H" >
  <match event="PO" condition="..."></match>
  <match event="POresponse" after="PO" condition="..."></match>
  <match event="POreceipt" after="POresponse" condition="..."></match>
</catch>
```

**NOTE:** Although it is possible to define such patterns as a combination of simpler "catch" operations correlated by a scriplet execution, the selection of event patterns inside a single catch operation allows for delegating such pattern detection to specialized, highly scalable event processors (e.g. CEP) if required by some implementations.

### 3.3.1 catch

**Role:** synchronizes the scriplet execution with an event or a pattern of events.

**XML Representation:**

```xml
<catch
  start ? = 'xsd:dateTime'
  tryfor ? = 'xsd:duration'
  eb ? = 'xsd:normalizedString'
  event-set ? = 'xsd:normalizedString'
  Content: match *, date-time ?
</catch>
```

• **catch/@eb:** identifies the event board that is source for the catching. If absent, the "main" event board is the default source.

• **catch/@tryfor:** a duration that indicates how long the catch operation may "try" to select an event (or a combination of) that satisfies the defined pattern. By default, tryfor=0, meaning the catch operator acts as an instant query on the event board.
**catch/match**: defines a condition for selecting one event (or a number of events) from the event board. See "match" detailed description.

**Execution semantics:**

**Step 1**: The first "match" child element will be executed first on the event board: it defines which event will be caught first. This is the "lead event" of the event pattern defined by catch. To be eligible for selection, a lead event must satisfy the following conditions:

- the lead event is not previously masked by any scriplet instance of the same package execution instance.
- the lead event posting time (dateTime) is set after the virtual present (VP-time). Note that the VP-time - when set before actual present time - is generally incremented by the execution of the catch, simulating time passing.

Once the lead event is selected, the VP-time is set to the posting time of this event.

**Step 2**: the next "match" statement (next "match" sibling) of the catch is executed. The event search for this event is no longer conditioned by the VP-time, i.e. could search for events before the lead event (past events) or after the lead event, depending on ordering constraints stated in the match e.g. as specified by the attribute match/@after. For this "match", the eligibility conditions are:

- the lead event is not previously masked by any scriplet instance of the same package execution instance.
- the event satisfies the selection condition for this match, which generally correlates with previously selected events. (selection conditions can only refer to events or variables defined in preceding "match" siblings).

In case the selected event(s) occurred after the VP-time known just before executing this step, the VP-time is set to the posting time of this event (or of last event of a set, see match/@min).

**Step n**: In case of three or more match statements, loop on "Step 2" until all match statements are executed.

**Time impact**: catch is a time-affecting operation, meaning it will modify the virtual present time (VP-time).

### 3.3.2 catch/match

**Role**: define an event profile and its place in an event pattern, for a catch operation.

The `<match>` element will select events based on a conditional expression specified in the `<condition>` child element. When a combination of two or more events needs be selected possibly based on different conditional expressions, each event may be named (here “E1” and “E2”), and a temporal order (possibly partial) may be specified between these using the attribute @after. This temporal order uses the event board posting time, not the originating time of the event.

**XML Representation:**

```xml
<match
    event ? = 'xsd:normalizedString'
    view ? = 'xsd:normalizedString'
    before ? = 'xsd:normalizedString'
    after ? = 'xsd:normalizedString'
    min ? = 'xsd:integer'>
    Content: condition *
</match>
```
• **catch/match/@event**: Gives a symbolic name to the event(s) selected. Usable as a variable later.

• **catch/match/@before**: list which other event(s) of the same pattern, must occur after this event.

• **catch/match/@after**: list which other event(s) of the same pattern, must have occurred prior to this event.

• **catch/match/condition**: The event selection condition (e.g. XPath expression).

• **catch/match/@view**: determines how much data from each selected event must be reproduced in the catch result. It is specified as a set of XPath expressions. If not present, the entire event is returned.

• **catch/match/@min**: determines the minimum number of matching events to select, in order to consider the match as successful. (default = 1)

Example 1: The following is a more complete example of a catch definition, that selects a sequence of three events representing an exchange of three messages:

(E1) a purchase order request,
(E2) a PO response (acceptance or rejection),
(E3) a generic receipt to the PO response

These three events are ordered in time: E3 after E2 after E1. The correlation between these events is based on:

* Between E1 and E2: the same value for the PO reference# ("order-ref" element here present in the header as a property)
* Between E2 and E3: the MessageID value present in E2, equal to RefMessageID in E3.

```
<catch>
  <match event="E1">
    <condition>
      >content/soap/Header/msgData/action = 'PORequest'</condition>
  </match>
  <match event="E2" after="E1">
    <condition>(content/soap/Header/msgData/action = "POAccept" or
               content/soap/Header/msgData/action = "POReject") and
               content/soap/Header/msgData/property[@name = 'order_ref'] =
               $E1/content/soap/Header/msgData/property[@name = 'order_ref']</condition>
  </match>
  <match event="E3" after="E2">
    <condition>
      >content/soap/Header/msgData/action = 'Receipt' and
      content/soap/Header/msgData/property/RefMessageID =
      $E2/content/soap/Header/msgData/messageID</condition>
  </match>
</catch>
```

Example 2: the temporal order ("after") used to order the events of a combination does not necessarily match the selection order of these events. Assume we want to select PO transactions where the PO was rejected. Because PO rejections are far less common than PO acceptances, we may decide, for performance reasons, to start selecting such combinations by selecting first the Rejection event (E2)
instead of the PO event (E1). This way the test engine will not have to process many irrelevant PO events. The catch will then be:

```xml
<catch>
  <match event="E2">
    <condition lg="xpath">
      content/soap/Header/msgData/action = "POReject"
    </condition>
  </match>

  <match event="E1" before="E2">
    <condition lg="xpath">
      content/soap/Header/msgData/action = "PORequest" and
      content/soap/Header/msgData/property[@name='order_ref'] =
      $E2/content/soap/Header/msgData/property[@name = 'order_ref']
    </condition>
  </match>

  <match event="E3" after="E2">
    <condition lg="xpath">
      content/soap/Header/msgData/action = 'Receipt' and
      content/soap/Header/msgData/property/RefMessageID =
      $E2/content/soap/Header/msgData/messageID
    </condition>
  </match>
</catch>
```

NOTE: the general rule when writing `<match>` elements in a catcher, is "no forward references". This means that in a `<match>` element, any variable referred to - such as "event variables" $E1 and $E2 - and any event referred to using "after" or "before", must have been set in a previous `<match>` element.

### 3.3.3 catch/date-time

**Role:** define ?

**XML Representation:**

```xml
<date-time
  start ? = 'xsd:date'
  time ? = 'xsd:time' />
```

**Effect of a Catch:**

The "effect" of a catch is the sequence of selected events (or views of these, in case the related match statements have defined a view).

### 3.3.4 mask

**Role:** mask an event or a pattern of events.
3.4 Conditional Statements

Conditional statements are of two kinds:
(a) simple conditions
(b) switch conditions

3.4.1 if

Role: define a simple condition

XML Representation:

```xml
<if expr='xsd:normalizedString'>
   Content: (call-adapter | catch | decide | eval | exit | if | loop | message | post | sleep | start | var | 'xsd:Any') *
   (this element also allows mixed content so text can be included directly as part of the main element's content)
</if>
```

Simple conditions are of the form:

```xml
<if expr="bool_expr1">
   body
</if>
```

The body of the `<if>` element is executed (in the same way as the body of a variable assignment) only if the condition is true. The body may be a sequence of one or more of the following: an eTSM statement or an XML fragment.

Effect of a simple condition:

The effect of a simple condition is evaluated as follows:

```plaintext
if (expr = true) then
   effect( body )
endif
```

3.4.2 decide

Role: define a Switch condition:

```xml
<decide>
   <if expr="bool_expr1">
      (body-1)
   </if>
</decide>
```
XML Representation:
<decide><Content: if, else-if *, else</Content></decide>

3.4.3 decide/if

XML Representation:
<if><expr = 'xsd:normalizedString'>
  <Content: (call-adapter | catch | decide | eval | exit | if | loop | message | post | sleep | start | var | 'xsd:Any') *
  {this element also allows mixed content so text can be included directly as part of the main element's content}</Content>
</if>

3.4.4 decide/else-if

XML Representation:
<else-if><expr = 'xsd:normalizedString'>
  <Content: (call-adapter | catch | decide | eval | exit | if | loop | message | post | sleep | start | var | 'xsd:Any') *
  {this element also allows mixed content so text can be included directly as part of the main element's content}</Content>
</else-if>

3.4.5 decide/else

XML Representation:
<else><Content: (call-adapter | catch | decide | eval | exit | if | loop | message | post | sleep | start | var | 'xsd:Any') *
  {this element also allows mixed content so text can be included directly as part of the main element's content}</Content>
</else>

Effect of a switch condition:

The effect of a switch condition is evaluated as follows (here illustrated on a switch with one "if", one "else-if", and one "else"):

```c
if ( bool_expr1 = true) then
  effect( 'body-1' )
else if ( bool_expr2 = true) then
  effect( 'body-2' )
else
  effect( 'body-last' )
```
3.5 Variables

3.5.1 var

Role: define a variable and assign a value.

XML Representation:

```xml
<var
   name   = 'xsd:normalizedString'
   type   = 'xsd:normalizedString'>
   Content: (call-adapter | catch | decide | eval | exit | if |
     loop | message | post | sleep | start | var | 'xsd:Any' ) *
   {this element also allows mixed content so text can be included directly as
    part of the main element's content}
</var>
```

Variables are typically assigned once within a scriplet: their visibility and usability scope is the scriplet. If scriplet "S1" invokes scriplet "S2" and needs to make a variable V1 usable by "S2", it uses parameters (V1 value is passed as parameter of S2).

A variable is assigned only once during its lifecycle.

A variable, like a parameter can be shadowed by subsequent declarations inside constructs contained by the scriplet.

After they have been initialized, variables are named in subsequent operations by prefixing their name with '$_', e.g. $myvar.

The @type attribute value must indicate the value type which can be:

- xml: the value of the variable is a serialized XML fragment.
- int: the value of the variable is an integer value.
- string: the value of the variable is a string value.

Variable assignment (or rather, initialization) is done in one of two ways:

(1) by enclosing a constant value inside the var element, e.g. a string value or a numeric value:

Examples:

The @type attribute must be set with an atomic type (int, string, real, double...) The 'string' type is the default type value.

```xml
<var name="greeting" type="string">hello</var>
```
(2) by enclosing an XML element (@type = 'xml').

Example 1:

```xml
<var name="PO" type="xml">
  <myapp:PurchaseOrder ref="1234">
    <myapp:Item>abc</myapp:Item>
    <myapp:totalamount>1200</myapp:totalamount>
  </myapp:PurchaseOrder>
</var>
```

This XML element may be an etsm-qualified statement or may contain etsm-qualified statements: these will automatically be interpreted before assignment. The semantics of assignment is that the "effect" of this XML element is assigned to the variable (in case there is no embedded etsm statement, the effect of an XML fragment is the fragment itself).

Any etsm qualified statement inside a variable declaration will be executed at the time the variable declaration is processed.

```xml
<var name="PO" type="xml">
  <etsm:catch>
    ...
  </etsm:catch>
</var>
```

As a general rule, the XML enclosure (XML child element(s)) of an <var> element is always processed, and its "effect" (see Section ...) is assigned to the variable. Processing a text node doesn't change its value - it is still a text node. When elements in the etsm namespace are processed, they may produce other nodes (ie text or elements). For other elements, the outer element and its attributes are copied, but whatever content it encloses is processed further. For example, the totalamount in the purchase order could be "parameterized" by including an <eval> element in the totalamount element.

Example 2:

```xml
<var name="PO">
  <myapp:PurchaseOrder ref="1234">
    <myapp:Item>abc</myapp:Item>
    <myapp:totalamount>
      <eval expr="$total"/>
    </myapp:totalamount>
  </myapp:PurchaseOrder>
</var>
```
The variable value in example 2 will be same as in Example 1 if the variable $total value is "1200".

Assigning the result of an expression - e.g. written using XPath - is done using the statement \texttt{etsm:eval}.

The following variable assignment will give $POamount the same value as the variable $total, when POamount is declared after PO in the previous example:

\begin{verbatim}
<var name="POamount" type="int">
  <eval type="XPath2" expr="$PO/myapp:PurchaseOrder/myapp:totalamount"/>
</var>
\end{verbatim}

The elements inside the purchase order wouldn't have to be limited to literal XML elements and \texttt{<eval>}. You could also use other instructions such as \texttt{<if>}:

\begin{verbatim}
<var name="PO">
  <myapp:PurchaseOrder ref="1234">
    <myapp:Item>abc</myapp:Item>
    <myapp:totalamount>
      <eval expr="$total"/>
    </myapp:totalamount>
    <if expr="$physicalItem">
      <myapp:weight>
        <eval expr="$weight"/>
      </myapp:weight>
    </if>
  </myapp:PurchaseOrder>
</var>
\end{verbatim}

\textbf{3.5.2 execution-context/var}

XML Representation:

\begin{verbatim}
<var
  name = 'xsd:normalizedString'
  type ? = 'xsd:normalizedString'>
  Content: xsd:normalizedString
</var>
\end{verbatim}

\textbf{Effect of a variable declaration:}

The effect of a variable declaration is null. Indeed, the effect of the XML element embedded in a variable definition is captured as the value of this variable, and this assignment in itself is not producing any effect. Effecting the value of this variable may be done by using it inside an eval statement: e.g. \texttt{<etsm:eval expr="$myvar"/>}

[see minutes Dec 15, 2009 meeting for all cases of var assignments]
3.6 The "Effect" of Various Constructs

The primary output of a scriplet execution is called an effect. The effect is a text output, which may be well-formed XML or just plain text.

By extension, the following artifacts are also said to have an effect:

- The effect of an XML fragment that does not contain any embedded etsm statement, is itself.
- The effect of an etsm:eval statement, is the result of the evaluation of the embedded expression.
- The effect of an XML fragment that contains some embedded etsm statement, is the same fragment after substituting the etsm statement with its effect.
- The effect of an etsm statement other than etsm:eval (e.g. <if>, <loop>, <start>...) is evaluated as specified in the definition of each statement.

3.6.1 In-line effects

The effect to be produced by a scriplet may be specified in-line, i.e. by inline inserts of externally-namespaced XML fragments inside the scriplet. An XML fragment is externally-namespaced if it has a different namespace from the etsm namespace, and if this namespace is declared in the execution context (<nms-binding>)

Because a scriplet must always be a well-formed XML unit in itself, the scriplet must also produce by itself a well-formed in-line effect, meaning that for each opening tag of an in-line effect fragment it must also specify the closing tag. It is not possible to specify an in-line effect opening tag in a scriplet, then the corresponding closing tag in another scriplet.

3.6.2 Inserting the value of expressions in effects: eval

XML Representation:

```
<eval
   lg ? = 'xsd:normalizedString'
   expr   = 'xsd:normalizedString'/>
```

... 

The eval statement evaluates an expression (e.g. XPath) and produces its value as effect. The result could be any string, including some XML fragment or partial XML fragment: unlike in-line effects, a scriplet using eval can produce the opening tag of an XML fragment while another scriplet will produce the closing tag.

3.6.3 Generating effect with CDATA

3.6.4 Concurrency Semantics

The combined effect of scriplets executing synchronously or asynchronously, is following these rules:

Rule 1. When scriplet S1 "starts" (synchronously or asynchronously) scriplet S2, the effect of S1 is obtained by:

1. obtaining the effect of S2 at the end of S2 execution,
(2) substituting the `<starts scriplet="S2" async="..." />` statement in S1 with the effect of S2.

(3) obtaining the effect of S1 at the end of its execution, after substitution in (2).

This rule applies recursively to all scriplet invocation trees, possibly combining synchronous and asynchronous invocations.

### 3.7 Iterations

#### 3.7.1 loop

XML Representation:

```
<loop>
  Content: on-start ? loopbody, next-val *, on-final ?
</loop>
```

```
loopbody = (catch | decide | eval | exit | if | loop | message | start | until | var | 'xsd:Any') *
```

Iterations - or loops - are represented by a single syntactic construct that covers both types of iterations:

* (a) Iterations over a collection of items returned by an expression or contained in a variable ("for-each")

* (b) Iterations that are controlled by a boolean expression, here until the expression is "true".

Loops in eTSM have a syntax that may appear a little heavier than in conventional programming languages. This is partly due to the restrictions on variable scopes and assignments, as well as to manage the "effect" of the loop.

It must be noted that more concise loops can be used inside expressions, e.g. "for / in" in XPath2 expressions that are defined for example in an eval statement, or in any statement that make use of an expression (e.g. match, until, filter).

Two kinds of variables may be defined inside a loop, that only differ in how they can be assigned or modified:

(a) **loop variable**: declared and initialized once in the `<on-start>` statement, such a variable keeps its value throughout the loop iterations, or it may be updated at the end of the loop body (`<next-val>` statement) based on its value from the previous iteration. These variables are also in scope of the `<on-final>` statement.

(b) **transient variable**: a transient variable is a regular variable declared in the usual way as done in scriplets. Defined in the loop body, it is only visible within the loop body. Its declaration and therefore its assignment is re-processed at each iteration, but unlike loop variables, its value cannot be set in one iteration for the next iteration run. Such variables are only in scope of the loop body, and not in scope of the `<on-start>` or `<on-final>` statement.

Besides these two types of variables that can be declared in the loop, other variables (including parameters) declared in the scriplet that defines the loop, are implicitly within scope and accessible by the loop body, although not mutable by it.

The loop element may contain the following child elements, in addition to regular etsm statements inside the loop body:
• An `<on-start>` element: any initialization operation that must be done just once before the loop iterates, is defined in this element. It must be the first child element of the `<loop>` element. This includes declaration of "loop variables".

• An `<on-final>` element: it contains operations that need be done once when the loop completes, either normally or as result of a break statement. If any aggregated result needs to be produced at the end of the iteration, this is where its computation is final. It must be the last child element of the `<loop>` element.

• An `<until>` statement. This defines the exit condition for the iteration. On execution, and if the condition is satisfied, the `<on-final>` element is executed. In such a case, the elements in the body at same level as `<until>` and after it are not executed (except for `<on-final>`). There MUST be at least one `<until>` statement in a loop.

• One or more `<next-val>` statements. These MUST always be at the end of the loop with no other statement after except for `<on-final>`. The `<next-val>` statement is re-assigning a loop variable, i.e. a var that must have been defined in the `<on-start>` statement. This re-assigning will be effective in the next iteration run, and may be based on the current values of any loop variable.

The loop body, is made of all statements inside the `<loop>` element, other than the above statements. The loop body may contain regular variable definitions and assignments (transient vars).

**Effect of a loop:**

The effect of a loop is the concatenation of the effect of the loop body at each iteration, followed by the effect of the `<on-final>` statement.

NOTE: the last iteration may exit (until) in the middle of the loopbody. In case the until statement is itself inside an in-line effect, this in-line effect will be produced well-formed. Indeed, the until statement semantics is to (a) prevent its following sibling statements from execution, (b) end the iterations at the end of the first execution of the loopbody where the until expression evaluates to true. The "effect" semantics of `<until>` is similar to the `<exit>` effect semantics.

The effect of the above loop is only the effect of the `<on-final>` statement, which is a single value (the value of the variable $pototal ). The loop body in this case does not produce any effect.

When embedding the following loop example in a variable definition, the value of this variable would be the final value of the variable $pototal at the time the loop completes.

**Example 1:**

```
The above example shows an iteration over the items of a purchase order (referred to by the $PO variable), in order to calculate the total amount ($pototal).

In the above example, both "loop" variables and "transient" variables are used:

- **Loop variables**: $myitems, $item, $pototal. The variable $myitems is used as is throughout the iterations. Only $item and $pototal are updated at each iteration, based on their value at the previous iteration.

- **Transient variables**: $myItemAmount, $itemRebate. The declaration of these is re-processed at each iteration, which entails a new value reassignment that ignores the previous value, and will be lost at the next iteration.

An example of use of a transient variable is for capturing a different event at each iteration run. In the variant below, the loop checks if there is a special sales period (represented by an event) for any of the PO items, and in that case uses the sales price instead of the catalog price:

**Example 2:**
<loop>
  <on-start>
    <loop-var name="myitems" expr="$PO/lineItems/item"/>
    <loop-var name="item" type="int">1</loop-var>
    <loop-var name="pototal" type="int">0</loop-var>
  </on-start>
  <until expr="$item fn:count($myitems)"/>
  <var name="itemSalesEvent">
    <catch>
      <match>
        <condition>content/sales/product[@ref = $myitem[$item]/reference]</condition>
      </match>
    </catch>
  </var>
  <var name="myItemPrice">
    <decide>
      <if expr="$itemSalesEvent">
        <eval expr="$itemSalesEvent/am:content/sales/product/salesprice"/>
      </if>
      <else>
        <eval expr="$myitems[$item]/unitprice"/>
      </else>
    </decide>
  </var>
  <var name="myItemAmount">
    <eval expr="$myItemPrice * $myitems[$item]/quantity"/>
  </var>
  <var name="itemRebate">
    <decide>
      <if expr="$myItemAmount gt 1000">
        <eval expr="$myItemAmount * 0.10"/>
      </if>
      <else>
        <eval expr="0"/>
      </else>
    </decide>
  </var>
  <next-val name="pototal" expr="$pototal + ($myItemAmount - $itemRebate) "/>
  <next-val name="item" expr="$item + 1"/>
  <on-final>
    <eval expr="$pototal"/>
  </on-final>
</loop>

3.7.2 loop/on-start

XML Representation:
<on-start>
3.7.3 loop/on-start/loop-var

XML Representation:

```xml
<loop-var
    name   = 'xsd:normalizedString'
    expr ? = 'xsd:normalizedString'
    type ? = 'xsd:normalizedString'>
    {this element also allows mixed content so text can be included directly as part of the main element's content}
</loop-var>
```

3.7.4 loop/next-val

XML Representation:

```xml
<next-val
    name   = 'xsd:normalizedString'
    expr ? = 'xsd:normalizedString'>
    {this element also allows mixed content so text can be included directly as part of the main element's content}
</next-val>
```

3.7.5 loop/on-final

XML Representation:

```xml
<on-final
    Content: (catch | decide | eval | if | loop | message | start | var | 'xsd:Any') *
    {this element also allows mixed content so text can be included directly as part of the main element's content}
</on-final>
```

3.8 Message Handling

3.8.1 message

XML Representation:

```xml
<message>
    Content: eval *
    {this element also allows mixed content so text can be included directly as part of the main element's content}
</message>
```
### 3.8.2 post

XML Representation:

```xml
<post
    message ? = 'xsd:string'
    event ? = 'xsd:normalizedString'>
    Content: (catch | decide | eval | exit | if | loop | message |
        sleep | start | var | 'xsd:Any' ) *
    {this element also allows mixed content so text can be included directly as
     part of the main element's content}
</post>
```

### 3.9 Exiting

The `<exit>` statement allows for interrupting the execution of a scriplet before its normal ending (execution of its final statement). The statement immediately causes the termination of the containing scriplet.

Such an interruption may propagate to the scriplet invoking the exiting scriplet, in case the `@bubble-exit` attribute was set to "true" in the `<start>` statement invoking the exiting scriplet. In this case it is said that the `<exit>` is "bubbling-up", i.e. causing the exiting of the invoking scriplet. By default, `<exit>` does not bubble-up to the invoking scriplet.

When the exit statement is used in a variable definition, it never bubbles-up beyond this variable definition to the scriplet where the variable is defined. Instead, it interrupts the execution inside the variable, and the resulting effect is simply assigned to the variable.

[Question#1]: finalize the actual effect of an exiting scriplet: two approaches:

1. the "normal" effect of the scriplet is discarded, and only the effect of the `<exit>` statement is produced.
2. the "normal" effect of the scriplet is preserved - just as for a loop that exits with `<until>` - i.e. any effect inside which the `<exit>` executed - like an in-line effect -, is still preserved. But the effect of `<exit>` is inserted in this effect. Rationale: exiting is not always used as an "exception" mechanism (abnormal termination). It is sometimes just a convenience for terminating a test case in various ways without having to use a cascade of conditional statements. In such cases, you want to preserve the effect built so far on top of `<exit>`.

### 3.9.1 exit

XML Representation:

```xml
<exit
    message ? = 'xsd:normalizedString'
    result ? = 'xsd:normalizedString'>
    Content: (call-adapter | catch | decide | eval | exit | if |
        loop | message | post | sleep | start | var | 'xsd:Any' ) *
    {this element also allows mixed content so text can be included directly as
     part of the main element's content}
</exit>
```
[Question#2]: decide what is the default for bubble-exit: Proposal: "false" should be the default instead of "true". Rationale: an invoking scriplet should use the invoked scriplet as a black box by default and not be affected by how it terminates.

In the following example, if S3 exits then S2 will still execute the condition that will produce the xml fragment <myapp:outcome>S3 exited</myapp:outcome>. In the presence of @bubble-exit="true" (@bubble-exit="false" by default) an exit in S3 would propagate to S2 and the conditional statement in S2 would not execute:

```xml
<scriplet name="S2">
    <start scriplet="S3" bubble-exit="true"/>
    <var name="result">...</var>
    <if expr="$result">
        <myapp:outcome>S3 terminated normally</myapp:outcome>
    </if>
</scriplet>

<scriplet name="S3">
    ...
    <if expr="">
        <exit/>
    </if>
    ...
</scriplet>
```

[Question#3]: When a scriplet exits, all its concurrent sub-invocations (scriplets that it has invoked in a concurrent mode) also terminate?

Proposal: when starting a scriplet S2 asynchronously, with bubble-exit="true", then the invoking scriplet will be interrupted whenever the invoked scriplet S2 is exiting, i.e. one of the following occurs:

(a) the statement in S1 executing while the exit occurs in S2, is a <catch>. In that case, the catch is aborting on failure, and S1 is exiting as if executing an <exit> statement just before the "catch".

(b) the statement in S1 executing while the exit occurs is other than "catch". In that case, it terminates properly, and S1 exits just after as if executing an <exit> statement just after.

If the exiting scriplet was creating another effect - or would have produced another effect without the exit - this effect is cancelled, and only the exit statement would appear in the resulting effect.

Effect of Exit:
The effect of <exit> is the effect of the contained XML fragment.

3.9.2 Exiting from Variable definitions

When the body of a variable definition "exits", the exiting does not bubble-up beyond the variable definition.

In the following, the body of variable v3 may exit, meaning the <catch> will not execute in such cases. But the test <if expr="$v3"> next in the scriplet body is always executed regardless whether the body of the variable v3 exited or not:

```xml
<scriplet name="S1">
    <var name="v3">
        <if expr="$v3 = 0">
            ...
        </if>
    </var>
</scriplet>
```
3.9.3 Exiting from Conditional Statements

By definition, an <exit> statement inside a conditional statement always appears either:

(a) as a leaf of a conditional tree, in which case the exiting is terminating the conditional execution, and also terminating the body of statements within which this conditional statement appears.

(b) as a statement in a sequence that is such a leaf. In that case, any statement in this sequence after the <exit> is not executing, and the conditional is terminating as described in (a).

3.10 Time Control

3.10.1 Updating the VP-time

The execution of a script package (starting with the execution of its "execution context") starts at a virtual date/time, the VP-time (virtual present time).

Every scriplet executes at a "starting time", which is:

- the VP-time at which the <start> statement is executing in the invoking scriplet - or the @datetime value if any, given in the <start> statement.
- the VP-time at which the initial scriplet executes: AP-time at which the script package executes, or @datetime value given as attribute of <start-with> argument, if any.

Every statement in a scriplet executes at a "starting time" which is the VP-time value just before executing the statement.

The VP-time is conditioning the event catching: typically, the "lead event" of a <catch> is the first event to satisfy the catch filters, posted after VP-time.

Two constructs can modify the VP-time:

1. **sleep**: this construct is blocking the scriplet execution thread for some duration.
   - If the date (VP-time + duration) is anterior to the actual present time (AP-time), <sleep> is simply incrementing VP-time with duration, and gives immediately control to the next statement.
   - If the date (VP-time + duration) is posterior to the actual present time (AP-time), <sleep> is waiting until AP-time = VP-time + duration, and gives control to the next statement.

2. **catch**: this construct is either acting as a query over the event board, or as a synchronizing statement (or both) based on the following rules:
   - If VP-time = AP-time, then <catch> is blocking the scriplet execution thread until some event(s) is caught that matches its <match> statement(s), or until the @tryfor duration is exhausted.
• If VP-time < (VP-time + tryfor) < AP-time (meaning both VP-time and (VP-time + tryfor) are anterior to AP-time), then the <catch> statement simply acts as a query over the event board. The VP-time is incremented up to the latest posting date/time of this event combination - if such an event(s) is caught before (VP-time + tryfor) - or is incremented up to VP-time + tryfor if no event satisfies the <catch> in the event board.

• If (VP-time < AP-time < (VP-time + tryfor)), then the <catch> statement starts as a query over the event board up to AP-time, then if no event(s) is selected, blocks the scriplet execution until some event(s) is caught that matches its <match> statement(s), or until the @tryfor duration is exhausted.

In all cases, the VP-time is moved to VP-time + @tryfor in case of catch failure, or to VP-time + datetime(event post) in case of successful catch.

3.10.2 sleep

XML Representation:

```xml
<sleep
    unit ? = 'xsd:normalizedString'
    until ? = 'xsd:dateTime'/>
```

3.10.3 Synchronous vs. Asynchronous Invocations

When a scriplet is invoked asynchronously (@async="true") the next statement in the invoking scriplet, will execute at a VP-time that is same as the VP-time of the <start> statement. In other words, both scriplets now execute concurrently according to the virtual present time.

When a scriplet is invoked synchronously (@async="false") the next statement (after <start>) in the invoking scriplet, will execute at a VP-time that is same as the VP-time set at the end of the invoked scriplet. In other words, the invoking scriplet blocked until the end of the invoked scriplet.

3.11 Adapters

3.11.1 call-adapter

XML Representation:

```xml
<call-adapter
    name ? = 'xsd:normalizedString'
    type ? = 'xsd:normalizedString'>
    Content: with-param *
</call-adapter>
```
3.11.2 call-adapter/with-param

XML Representation:

```xml
<with-param
    name   = 'xsd:normalizedString'
    expr   = 'xsd:normalizedString'/>
```

...

3.12 Functions

The primary objective of functions is advanced computations that may exceed the capability of eTSM scripting. Functions can be invoked from many other statements including from inside expressions (e.g. an XPath expression), unlike scriplets. They may also be written in "foreign" languages or scripts, e.g. in Java, C# or XSLT, making it possible to reuse advanced function libraries and operators available for these languages, while scriplets are exclusively coded in eTSM syntax.

A function returns a value that could be an XML fragment. This value may be typed, and by default is a string possibly empty.

The following is an example of function returning an XML fragment of root <PO>. The value of child elements <POReference>, <Customer> and <Amount> are resulting from the evaluation of expressions, here simply returning the value of parameters passed when invoking the function:

```xml
<function name="purchaseorder">
    <param name="POref"/>
    <param name="cust"/>
    <param name="amount"/>
    <myapp:PO>
        <POReference><eval expr="POref"/></POReference>
        <Customer><eval expr="$cust"/></Customer>
        <Amount><eval expr="$amount"/></Amount>
    </myapp:PO>
</function>
```

XML Representation:

```xml
<function
    name   = 'xsd:normalizedString'>
    Content: param *, (decide | eval | exit | if | message | var | 'xsd:Any')*
    {this element also allows mixed content so text can be included directly as part of the main element's content}
</function>
```

...

The child elements of 'function' are, with some exceptions, the same as those of the same name in 'scriplet', above. There are elements allowed in scriplets which are not allowed in functions. Elements of functions are as follows.
3.12.1 decide

XML Representation:

```xml
<decide>
  Content: if, else-if *, else
</decide>
```

3.12.2 decide/if

XML Representation:

```xml
<if
  expr = 'xsd:normalizedString'>
  Content: (decide | eval | exit | if | message | var | 'xsd:Any' ) *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</if>
```

3.12.3 decide/else-if

XML Representation:

```xml
<else-if
  expr = 'xsd:normalizedString'>
  Content: (decide | eval | exit | if | message | var | 'xsd:Any' ) *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</else-if>
```

3.12.4 decide/else

XML Representation:

```xml
<else>
  Content: (decide | eval | exit | if | message | var | 'xsd:Any' ) *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</else>
```

3.12.5 exit

XML Representation:

```xml
<exit
  message ? = 'xsd:normalizedString'
  result ? = 'xsd:normalizedString'>
  Content: (decide | eval | exit | if | message | var | 'xsd:Any' ) *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</exit>
```

3.12.6 if

XML Representation:

```xml
;if
  expr = 'xsd:normalizedString'>
  Content: (decide | eval | exit | if | message | var | 'xsd:Any' ) *
  {this element also allows mixed content so text can be included directly as
part of the main element's content}
</if>

### 3.12.7 var

**XML Representation:**

```xml
<var
  name = 'xsd:normalizedString'
  type ? = 'xsd:normalizedString'>
  Content: (decide | eval | exit | if | message | var | 'xsd:Any') *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</var>
```

The following are no different when child elements of functions as when child elements of scriplets: `<param>`, `<eval>` and `<message>`.

Another difference between scriplets and functions, is that the effect of executing a function is all in its returned result. There is no other side-effect observable in eTSM (besides any intentional event posting done by the function).

In contrast, a scriplet does not explicitly return a result. However, its execution produces an "effect" - analogous to a side-effect - which is the concatenation of the effect of each step inside it. Thus, a scriplet execution could compose an XML fragment as its effect. Assigning a scriplet execution to an eTSM variable, amounts to assigning the effect of its execution as value of the variable.

In the above examples, the effect of executing the validatePOresponse scriplet, is produce by the execution of `<exit>` statements. This statement produces an XML fragment as effect, that can be assigned to a variable. This variable can then be used to generate a status event or other purpose.

So if the above scriplet is invoked from another scriplet in the following way, the variable `$POstatus" would contain the effect of any of the `<exit>` statements generated by the validatePOresponse scriplet:

```xml
<scriplet name="logPOStatus">
  <var name="myPO">
    <catch>
      <match><condition>emam:content/soap:Envelope/soap:Header/messageData[
        action = 'PORequest']</condition></match>
    </catch>
  </var>
  <var name="POstatus">
    <start scriplet="validatePOresponse">
      <with-param name="myPO" expr="$myPO"/>
      <with-param name="mesgId" expr="$myPO/emam:content/soap:Envelope/soap:Header/messageInfo/MessageId"/>
    </start>
  </var>
  <post event="$POstatus"/>
</scriplet>
```
4 The Base Language Subset

(this is the subset of the script language that is implementable with XSLT2.0)

4.1 The Script-package

4.1.1 script-package

XML Representation:
<script-package
  name ? = 'xsd:normalizedString'
  schemaVersionId ? = 'xsd:normalizedString'>
  Content: execution-context ?, (scriplet | function ) *
</script-package>

4.1.2 execution-context

XML Representation:
<execution-context>
  Content: nms-binding *, start-with , var *
</execution-context>

4.1.3 nms-binding

XML Representation:
<nms-binding
  prefix   = 'xsd:normalizedString'>
  Content: 'xsd:anyURI'
</nms-binding>

4.1.4 start-with

XML Representation:
<start-with
  scriplet   = 'xsd:normalizedString'/>

4.1.5 condition

XML Representation:
<condition
4.1.6 execution-context/var

XML Representation:

```xml
<var
   name   = 'xsd:normalizedString'
   type    = 'xsd:normalizedString'>
   Content: xsd:normalizedString
   {this element also allows mixed content so text can be included directly as part of the main element's content}
</var>
```

4.1.7 Scriplets

4.1.8 scriplet

XML Representation:

```xml
<scriptlet
   name   = 'xsd:normalizedString'>
   Content: param *, (catch | decide | eval | exit | if | loop | message | start | var | 'xsd:Any') *
   {this element also allows mixed content so text can be included directly as part of the main element's content}
</scriptlet>
```

4.1.9 param

XML Representation:

```xml
<param
   name   = 'xsd:normalizedString'
   type     = 'xsd:normalizedString'/>  
```

4.1.10 catch

XML Representation:

```xml
<catch
   following   = 'xsd:normalizedString'
   start   = 'xsd:dateTime'
   try-duration   = 'xsd:integer'>
   Content: match
</catch>
```
4.1.11 catch/match

XML Representation:

```
<match>
  Content: condition ?
</match>
```

4.1.12 condition

XML Representation:

```
<condition>
  Content: 'xsd:normalizedString'
</condition>
```

4.1.13 decide

XML Representation:

```
<decide>
  Content: if, else-if *, else
</decide>
```

4.1.14 decide/if

XML Representation:

```
<if
  expr = 'xsd:normalizedString'>
  Content: (catch | decide | eval | exit | if | loop | message | start | var | 'xsd:Any' ) *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</if>
```

4.1.15 decide/else-if

XML Representation:

```
<else-if
  expr = 'xsd:normalizedString'>
  Content: (catch | decide | eval | exit | if | loop | message | start | var | 'xsd:Any' ) *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</else-if>
```
4.1.16 decide/else

XML Representation:
<else>
  Content: (catch | decide | eval | exit | if | loop | message | start | var | 'xsd:Any') *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</else>

4.1.17 eval

XML Representation:
<eval
  expr = 'xsd:normalizedString'/>

4.1.18 exit

XML Representation:
<exit
  message ? = 'xsd:normalizedString'
  Content: (catch | decide | eval | exit | if | loop | message | start | var | 'xsd:Any') *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</exit>

4.1.19 if

XML Representation:
<if
  expr = 'xsd:normalizedString'>
  Content: (catch | decide | eval | exit | if | loop | message | start | var | 'xsd:Any') *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</if>

4.1.20 loop

XML Representation:
<loop
  Content: on-start ? (catch | decide | eval | exit | if | loop | message | start | until | var | 'xsd:Any') *, next-val *, on-final ?
</loop>
4.1.21 loop/on-start

XML Representation:

```xml
<on-start>
  Content: loop-var *
</on-start>
```

4.1.22 loop/on-start/loop-var

XML Representation:

```xml
<loop-var
  name = 'xsd:normalizedString'
  expr ? = 'xsd:normalizedString'
  type ? = 'xsd:normalizedString'>
  Content: xsd:normalizedString
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</loop-var>
```

4.1.23 loop/next-val

XML Representation:

```xml
<next-val
  name = 'xsd:normalizedString'
  expr ? = 'xsd:normalizedString'>
  Content: xsd:normalizedString
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</next-val>
```

4.1.24 loop/on-final

XML Representation:

```xml
<on-final>
  Content: (catch | decide | eval | if | loop | message | start |
            var | 'xsd:Any' ) *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</on-final>
```

4.1.25 message

XML Representation:

```xml
<message>
  Content: eval *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</message>
```
4.1.26 start

XML Representation:

```xml
<start
    scriptlet   = 'xsd:normalizedString'
    async ? = 'xsd:boolean'
    bubble-exit ? = 'xsd:boolean'>
    Content: with-param *
</start>
```

4.1.27 var

XML Representation:

```xml
<var
    name   = 'xsd:normalizedString'
    type ? = 'xsd:normalizedString'>
    Content: (catch | decide | eval | exit | if |
              loop | message | start | var | 'xsd:Any' ) *
    {this element also allows mixed content so text can be included directly as
     part of the main element's content}
</var>
```

4.2 Functions

4.2.1 function

XML Representation:

```xml
<function
    name   = 'xsd:normalizedString'>
    Content: param *, (decide | eval | exit | if | message | var | 'xsd:Any' ) *
    {this element also allows mixed content so text can be included directly as
     part of the main element's content}
</function>
```

4.2.2 decide

XML Representation:

```xml
<decide>
    Content: if, else-if *, else
</decide>
```

4.2.3 decide/if

XML Representation:
<if
  expr = 'xsd:normalizedString'>
  Content: (decide | eval | exit | if | message | var | 'xsd:Any' ) *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</if>

### 4.2.4 decide/else-if

XML Representation:
<else-if
  expr = 'xsd:normalizedString'>
  Content: (decide | eval | exit | if | message | var | 'xsd:Any' ) *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</else-if>

### 4.2.5 decide/else

XML Representation:
<else>
  Content: (decide | eval | exit | if | message | var | 'xsd:Any' ) *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</else>

### 4.2.6 exit

XML Representation:
<exit
  message ? = 'xsd:normalizedString'>
  Content: (decide | eval | exit | if | message | var | 'xsd:Any' ) *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</exit>

### 4.2.7 if

XML Representation:
<if
  expr = 'xsd:normalizedString'>
  Content: (decide | eval | exit | if | message | var | 'xsd:Any' ) *
  {this element also allows mixed content so text can be included directly as part of the main element's content}
</if>
4.2.8 var

XML Representation:

```xml
<var
    name   = 'xsd:normalizedString'
    type   = 'xsd:normalizedString'>
    Content: (decide | eval | exit | if | message | var | 'xsd:Any') *
    {this element also allows mixed content so text can be included directly as part of the main element's content}
</var>
```

The following are no different when child elements of functions as when child elements of scriplets: <param>, <eval> and <message>. 
5 Event Model and Management

5.1 Event Structure

The event structure is defined by the following schema:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"

targetNamespace="http://docs.oasis-open.org/tamie/emam/draft/20090909" xmlns="http://docs.oasis-open.org/tamie/emam/draft/20090909" version="0.4">
  <xs:element name="event-board" type="event-board_type"/>
  <xs:element name="event" type="event_type"/>
  <xs:complexType name="content_type" mixed="true">
    <xs:sequence>
      <xs:any namespace="##any" processContents="skip" maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:normalizedString"/>
    <xs:attribute name="schema" type="xs:normalizedString"/>
  </xs:complexType>
  <xs:complexType name="event-board_type">
    <xs:sequence>
      <xs:element name="events" type="events_type"/>
    </xs:sequence>
    <xs:attribute name="schema-version-id" type="xs:normalizedString"/>
    <xs:attribute name="id" type="xs:normalizedString"/>
    <xs:attribute name="timestamp" type="xs:dateTime"/>
  </xs:complexType>
  <xs:complexType name="event_type">
    <xs:sequence>
      <xs:element name="content" type="content_type" minOccurs="0"/>
    </xs:sequence>
    <xs:element name="event-properties" type="event-properties_type" minOccurs="0"/>
    <xs:attribute name="id" type="xs:normalizedString"/>
    <xs:attribute name="post-time" type="xs:dateTime"/>
  </xs:complexType>
</xs:schema>
```
5.2 Event Adapters

Event adapters are converting external events into eTSM events, or vice-versa.

An Event Adapter can be a direct source to an Event Board, or can be a source to an event queue - or store - that is itself a source to an Event Board.

5.3 Event Boards

An event Board is an event store that is directly accessible by scripts. An event Board is queryable, i.e. allows for selection and correlation of past events. An Event Board may be virtual, i.e. defined as a "view" over a physical event store (in a similar way views may be created over a database).

An eTSM script may use one or several Event boards. For example, a script may deal with three event boards:

(a) a read-only event board representing a log of events to be analyzed,
(b) a temporary event board for the sake of internal synchronization of the script execution, i.e. for which the script has read-write access,

(c) an output event board to record the results of the analysis of the event board (a).

A temporary event board is intended to be periodically flushed, e.g. between two executions of a scriplet. More precisely, one may want to give a temporary event board the semantics of a stack: all temporary events posted by a scriplet invocation will be flushed at the termination of this scriplet. Event boards are identified by URIs but assigned symbolic names for use in the scriplets, in a similar way XML namespace prefixes are associated with a formal URI.

A scriplet may be written with its statements only using symbolic names for its Event Boards (e.g. when posting events or catching events). The binding to actual event boards will then depend on the invoking scriplet.

### 5.3.1 Access State of an Event Board

The "access state" of an Event Board for a given Script determines the events that are accessible by a Script at some given time. The Access state does not depend uniquely on the Event Board itself, but also on the script that access it. There is then an Access State for each pair (Event Board, Script).

The part of the Access state that depends on the Script (i.e. may change from one Script to another, although these access the same Event Board) is:

- Virtual Present time (VPT), which is specific to the execution of a Script instance.
- masked events, which makes events invisible to a particular Script instance.

Three operations affect the Access state of Event Boards:

**Catch:** this operation is used by scripts to query past events or to wait for future events.

**Post:**

**Mask:** Event masking: Once a `<catch>` statement has captured some event(s), it may be necessary to mask these events so that they will not be caught again by subsequent `<catch>` operations.

This masking is scoped to a particular script (and its invocation closure).

### 5.4 Event Manager
6 Conformance
Appendix A. Glossary

Live Execution

Test Case

A set of test tools, software or files (data, programs, scripts, or instructions for manual operations) that verifies the adherence of a test assertion target to one or more normative statements in the specification. Typically a test case is derived from one or more test assertions. Each test case includes: (1) a description of the test purpose (what is being tested - the conditions / requirements / capabilities which are to be addressed by a particular test), (2) the pass/fail criteria, (3) traceability information to the verified normative statements, either as a reference to a test assertion, or as a direct reference to the normative statement.
Appendix B. References

[CONF1] Conformance requirements for Specifications (OASIS, March 2002)
see http://www.oasis-open.org/committees/download.php/305/conformance_requirements-v1.pdf

[CONF2] Conformance testing and Certification Framework (OASIS, Conformance TC, June 2001)

[TD] Test Development FAQ, WG note (W3C, 2005)
see http://www.w3.org/QA/WG/2005/01/test-faq

see http://www.w3.org/TR/2005/NOTE-spec-variability-20050831/

see http://www.w3.org/TR/2005/NOTE-test-metadata-20050914/
Appendix C. Related Material
Appendix D. Acknowledgments

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

Participants:
  •