Event-driven Test Scripting Language

Working Draft 0.85, 7 November 2007

Document identifier:

etsl-wd-085

Location:

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

Technical Committee:

OASIS ebXML IIC TC

Editors:

Jacques Durand, Fujitsu Computer Systems <jdurand@us.fujitsu.com>

Contributors:

Hyunbo Cho, KorBIT
Michael Kass, NIST
Jungyub Woo, Samsung Electronic
Monica Martin, Sun
Serm Kulvataniou, NIST
Jaewook Kim, NIST
Pete Wenzel, Sun

Acknowledgments

The following persons have provided valuable feedback and requirement material to this work:

Kenji Nagahashi, Fujitsu Laboratories
Nenad Ivezic, NIST

Abstract:

The Event-driven Test Scripting Language (eTSL) is a model and language for eBusiness / eGovernment test suites, with a particular focus on communication events, versatile usage for the QA testing phase as well as the monitoring of deployed systems, and extensible design to leverage

Copyright © OASIS Open 2006. All Rights Reserved.

specialized validation processors as well as XML tools such as Xpath [11], XSLT[1] and Xquery[12]. It builds on previous experiences in conformance and interoperability testing in ebXML, WS-I as well as on learning from eBusiness proof of concepts and testing environments done within the context of the OAGI-NIST testbed [] and the KorBIT testbed [].

Status:

This is a Working Draft, meaning that the TC has not necessarily reached consensus on any or all content, and all contents are subject to change.

Table of Contents

1.Introduction..3
2.Requirements and Related Work...................................5
  2.1 The Value of Test Case Scripting.............................5
  2.2 Related Work...6
3.Test Case Execution Model..9
  3.1 Main Concepts..9
  3.2 Deferred Testing vs Live Testing............................12
4.The Event Board...15
  4.1 The Event Model...15
  4.2 Event Board Operations......................................16
5.Test Case Scripting and Semantics...............................20
  5.1 Conventions...20
  5.2 Main Test Case Constructs...................................20
  5.3 Scripting of Test Steps.....................................28
6.Use Cases...35
  6.1 Use Case 1: Business transaction monitoring.................35
  6.2 Use Case 2: Scripting a Test Suite..........................39
  6.3 Use Case 3: Combined validation of Business Document and Transaction...42
  6.4 Use Case 4: Selective Reliability for Some Business Transactions..42
  6.5 Additional Use Cases..43
7.Operational Requirements..45
8.References..46
9.Appendix: Notices...48

Copyright © OASIS Open 2006. All Rights Reserved.

1.Introduction

Test cases for e-Business / e-Government will be used to verify the conformance of a partner endpoint to the various standards and conventions that communicating partners have agreed on. These can be of different nature:

- Messaging infrastructure standards, ranging from transport level - e.g. HTTP [21], SMTP [22] - to higher-level messaging protocols and quality of service (reliability, security), such as those defined as SOAP extensions.
- Message choreographies. These include business choreographies (e.g. Conforming to UMM business transaction patterns [3]) as well as lower level exchange patterns that are often necessary to establish some quality of service requirements and related context (reliability, trust, security).
- Business document standards, industry-specific such as XML schemas, or horizontal (e.g. Modeling practice). These include syntactic as well as semantic conformance. Also "metadata documents" such as agreements, are included in this category.

There have been conformance and interoperability testing tools and initiatives for each one of these categories, but there is currently no support for testing an *integration* of the above. WS-I profiles [17] address mainly the integration of standards within the messaging infrastructure layer, and remain horizontal in nature (industry-independent).

In practice, a deployed e-Business system will profile such an integration, involving all above layers in addition to industry-specific profiles. Relying on testing each layer independently will not be sufficient to test this integration. Testing for the conformance (or interoperability) of the sum is more than the sum of testing for the conformance of its parts.

For example, a user community may decide that a particular messaging QoS (reliability and security levels) should be used for specific business transactions but not for others. Such requirements assume an ability to generate and verify test cases that mix low-level message controls and business-level transactions aspects (e.g. involving the use of specific business documents in payloads.) Or, some business transaction patterns must bind to lower-level protocols in a very specific way, e.g. regarding the use of the HTTP back-channel, depending on how soon a business response is expected, which is a business attribute.

Such conventions cross all layers of an eBusiness stack. An other way to look at it, is to consider such conventions as a user-defined profiling of combined infrastructure standards. Such profiles are composed of:

(1) A set of horizontal infrastructure standards (messaging protocols and QoS, business transaction definitions, document modeling, as well as integration profiles such as WS-I). Some options allowed by these standards may explicitly be prohibited or mandated.

(2) A layer of user-defined or industry-specific standards on top of the above (business-specific header content and extensions for messaging, specific business transactions and related rules, document schemas and business rules).

Again, layers (1) and (2) above are depending on each other, and call for an integrated approach to conformance and interoperability, e.g. some option in (1) may be required for a particular business document or transaction defined in (2), but not for others.

The objective of this document is to define a model for such integrated testing, and a scripting

Copyright © OASIS Open 2006. All Rights Reserved.

representation for the related test cases. It builds on previous experiences in conformance and interoperability testing in ebXML [13], WS-I [18] as well as on learning from past e-Business proof of concepts for business collaboration done with OAGI material, in particular within the context of the OAG-NIST testbed and semantic document validation.

Copyright © OASIS Open 2006. All Rights Reserved.

2.Requirements and Related Work

2.1 The Value of Test Case Scripting

Why should the scripting of test cases be standardized? There are two requirements motivating 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.

The test case model and scripting defined in this document must be usable in two different contexts, both for conformance and interoperability testing:

- As input to a test Driver, that simulates a business endpoint. The objective is either to test the conformance of a remote eBusiness endpoint to the standards and profiles adopted by a user community, or to make an interoperability assessment.
- Monitoring of in-production e-Business systems. The objective is to verify the conformance of actual business transactions to their specifications, or to troubleshoot interoperability problems.

There are four fundamental requirements for the eTSL:

Simplicity. eTSL implementations must be easy to develop and validate. Also, test cases must be easy to understand by humans. This calls for a minimal set of control features sufficient for e-business test cases of modest size. For this reason, the script language will not use the full range of conventional workflow operators, e.g. as found in [15]. A simpler scripting also reduces the variations across platforms that are inevitable with rich scripting or programing languages. In a testing context, these variations in the interpretation of a test case script would be a serious liability.

XML-based. The test script and expression must rely on XML syntaxes and related technologies such as XPath [11] and XQuery [12]. This allows the scripting to work naturally with most of today's e-Business specifications, which are XML-based. Even if a message protocol is not XML-based, an event-adapter can wrap it into an XML format. By relying on XML tools and markups that are platform-ubiquitous, one also ensures the portability of the test scripting, model and engine.

Event-centric and time-independent execution model. The same test script must be executable either for real-time verification or as off-line (deferred) validation over a log of the interaction. a deferred or live-validation context. Test cases also must be able to react to all sorts of events, and correlate past events. For these reasons, all input must be captured in the form of events and

Copyright © OASIS Open 2006. All Rights Reserved.

wrapped into a standard event (XML) envelope. This is in contrast with the IIC test framework [10] where only business messages were represented as events. The coordination of test-case executions within a test suite is also event-driven. The state of the test case workflow (e.g. variables, last step executed) is also represented as events so that no additional persistence mechanism is required by a recoverable test engine.

Protocol-agnostic and platform-ubiquitous: Test script logic and control are abstracted from e-business protocols; it is versatile for messaging, business process, and business content testing regardless of technologies. Hence it can be used with either ebXML [13] AS2 or Web Services message profiles []. Of course a test case script that verifies business headers in ebXML may not apply to Web service messages, but a change in event-adapter should be the only modification needed to adapt a test script focused on, say, verifying business transaction and payloads, from one message protocol to the other.

2.2 Related Work

A few scripting or coding solutions for representing executable test cases have been proposed. They are presented below, and compared with the eTSL orientation.

ATML (Automatic Test Mark-up Language)

In its requirements, this specification [18 will provide XML Schemata and support information that allow the exchange of diagnostic information between conforming software components applications. The overall goal is to support loosely coupled open architectures that permit the use of advanced diagnostic reasoning and analytical applications. The objectives are therefore different than eTSL objectives, as ATML is focusing on the representation and transfer of test artifacts: diagnostics, test configuration, test description, instruments, etc. The focus of ATML is not on the scripting and execution model for test cases. ATML representations could then be complementary or auxiliary to test suites expressed in eTSL.

TTCN-3: Testing and Test Control Notation V3

TTCN-3 [6] is a powerful, computationally complete procedural language for expressing test cases, with an important user community. It has been used in and primarily inspired from the telecommunication domain.
In contrast, the set of algorithmic controls and datatypes is deliberately minimal in eTSL, as it does not aim at completeness, but at covering the most common test case patterns as observed in eBusiness / eGovernment areas, and as specified by existing mark-ups or models governing such exchanges (UMM, ebBP[7], WS-BPEL[9]). eTSL also aims at ease of implementation, delegating specialized processing to external calls (e.g. semantic validation of business documents), and relying on a few simple controls, even if that makes the scripting of some test cases less elegant. There are however two types of artifacts most important when validating business transactions - events and XML documents - for which eTSL provides better support. Although TTCN-3 has a notion of event (communication and timer events) it lacks support for handling complex events that are business objects, as well as an event board structure that can manage past events search and correlation. TTCN-3 leverage of XML is mostly reduced to its use as an option for its control interface TCI, while eTSL makes it a central scripting dialect, both for the objects it manipulates (for a large part XML documents) and for the scripting of test cases.
However, eTSL could be implemented on top of TTCN-3 extended with external XML processing and event board management: this would support test cases that integrates the lower layers of the protocol.

Copyright © OASIS Open 2006. All Rights Reserved.

EbXML TestFramework 1.1

eTSL improves on IIC past work (TestFramework 1.1 [10]) by addressing the feedback of implementors and users summarized below:

(a)- Easier extensions and integration of business-level testing modules (e.g. semantic tests for business document content, as required by NIST).
(b)- More versatile notion of event than the one supported by the former "message store". The message store is replaced by an “Event Board”. The execution of test cases may be deferred and validate past business transactions by analyzing past events.
(c)- Design is not ebXML centric (easier to use in other contexts than ebXML) (required by: NIST, RosettaNet[16][19], KorBIT).
(d)- Better support for business transaction testing, e.g. UMM. Conformance testing and monitoring of message choreographies, as expressed by ebBP or WS-BPEL. (required by: BPSS [7] and business transaction users)
(e)- The script language is simplified from the previous IIC (TestFramework 1.1) version, with a minimal set of workflow operations, in order to make implementations easier.
(f)- Various improvements suggested by implementors such as KorBIT.

JXUnit and JXU

JXUnit [4] and JXU [5] is a general scripting system (XML based) for defining test suites and test cases aimed at software testing. Test steps are written as Java classes. There is neither built-in support for e-Business testing nor support for the event-driven features that are central to eTSL. However, as a general test scripting platform that relies on a common programming language, this system could be used as an implementation platform for eTSL.

Choreography Languages

These are standards for specifying the orchestration of business processes and/or transactional collaborations between partners. Although a markup like XPDL [15] is very complete from a process definition and control viewpoint, it is lacking the event-centric design and event correlation / querying capability required by testing and monitoring exchanges. Also, a design choice has been here to use a very restricted set of control primitives, easy to implement and validate, sufficient for test cases of modest size.
Other languages or mark-ups define somehow choreographies of messages and their properties: ebBP (ebXML), WSDL, WS-BPEL, WS-Choreography[8]. The general focus of these dialects is either the operational aspect of driving business process or business transactions (e.g. WS-BPEL), and/or the contractual aspect – interface of transaction specification - (e.g. ebBP, WSDL) but not monitoring and validation.

Could they be used for scripting test cases about the very choreographies they define? This would introduce unwanted dependencies between the units under test and the test environment, as the implementations of these languages are precisely the targets of monitoring and validation. In addition, they have functional limitations w/r to testing and validation. Although they may express detailed conformance requirements, they fall short of covering the various aspects of an exhaustive conformance check e.g. the generation of intentional errors or simulation of uncommon behaviors. In addition, the focus of these languages is mainly on one layer of the choreography – they for instance ignore lower-level message exchanges entailed by quality of service concerns such as reliability, or binding patterns with the transport layer. As a result, an integrated testing of the behavior of their implementations in combination with expected bindings and QoS, is still to be supported.

Copyright © OASIS Open 2006. All Rights Reserved.

As an example of functional mismatch on a more detailed level, event processing when supported in above languages puts the emphasis on event-as-control (event handlers, “pick”, and run-time use of correlation sets in WS-BPEL), not as analysis material for conformance or interoperability assessment. By contrast, eTSL uses event correlation for analyzing past activities, and for expressing integrity rules on business transactions that require Xpath-based querying and timing control.

Copyright © OASIS Open 2006. All Rights Reserved.

3.Test Case Execution Model

3.1 Main Concepts

This section describes the basic concepts that support the test objectives outlined in the requirements section. These concepts turn into high-level primitives of eTSL, which allow for concise expression of test cases. These functions include:

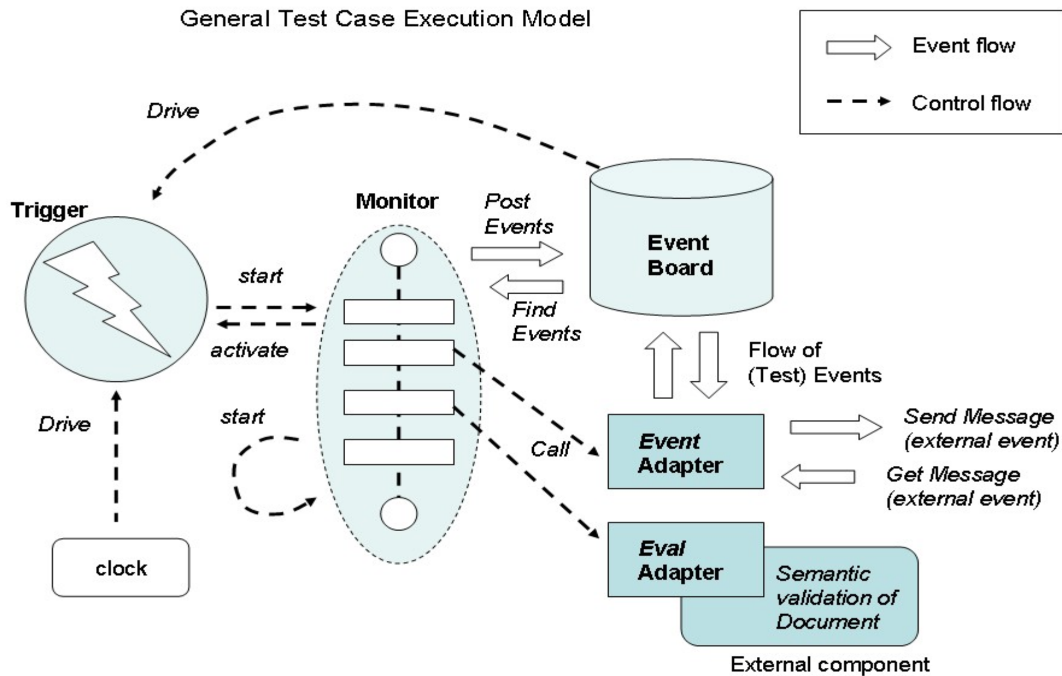
- Workflow control based on a thread model,. This is embedded in the notion of **Monitor**, which is the basic execution unit for test cases.
- Event-driven scripts. The general control of test case execution within a test suite, and of the test suite itself is represented by **Triggers** which define under which conditions and events an execution takes place.
- Event logging and correlation. Event management, central to eTSL, is supported by an entity called **Event Board**. The Event Board normally suffices for mediating all inputs to a test case, as well as outputs.
- Messaging gateways. Message traffic expected in all e-Business applications, is mapped to and from events. **Event-Adapters** perform these mappings, allowing for abstracting test cases from communication protocol aspects.
- Semantic test plug-ins. Advanced verifications on business documents, ranging from schema validation to semantic rules over business content, are delegated to **Eval-Adapters**.

While these features may themselves be potentially complex, it has been possible in eTSL - based on previous experience - to identify a minimal set of controls sufficient for e-Business testing. For example, workflow control only makes use of the simplest control primitives that have proved sufficient for test cases, not pretending to replicate the full range of workflow operators. Event correlation and querying rely on simple selection expressions based on XPath. Also, business document validation that goes beyond simple XPath-based checks on XML documents , is delegated to external plug-ins.

The overall model of test case execution is illustrated by the following figure.

Figure 1. Test Case Execution Model

Copyright © OASIS Open 2006. All Rights Reserved.



The main components are:

Monitor: Represents the logic of a test case. A test case may use several monitors in its definition, and a test case instance may engage the concurrent or sequential execution of several monitor instances. A monitor is a script that specifies the steps and workflow of the test case. A monitor instance is always created as the result of a start operation executed either by another monitor or by a ***trigger***. The first monitor started for a test case (i.e. Started by a Trigger) is called ***root monitor*** for the test case. There is always a trigger at the origin of monitor(s) execution (directly or indirectly). A monitor instance can start another monitor instance concurrently to its own execution, and can activate another trigger. The outcome of a test case (pass / fail / undetermined) is determined by the final outcome of the monitor(s) implementing this test case. The execution of a monitor produces a ***trace*** that can be posted as an event.

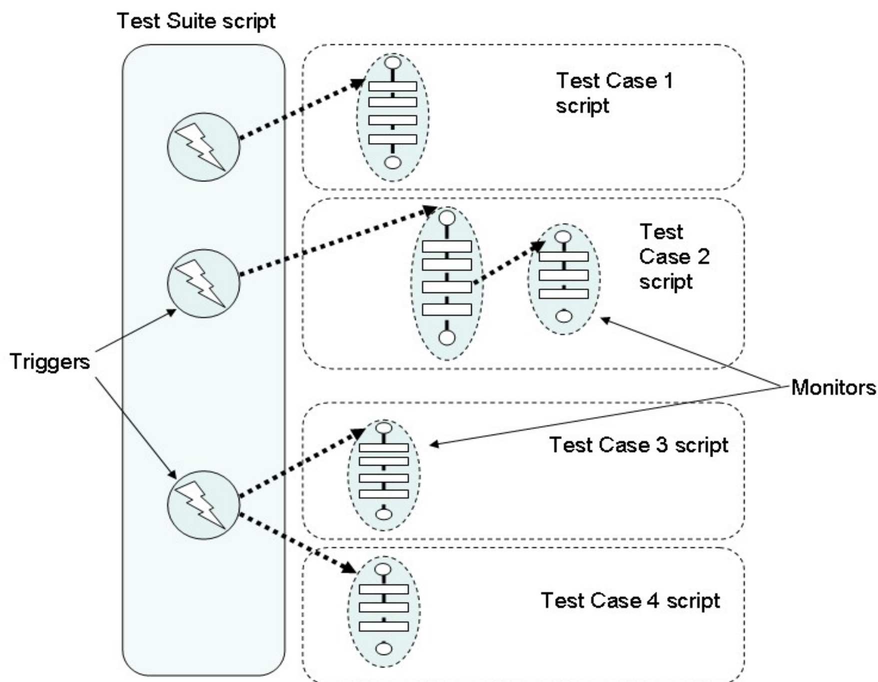
Trigger: The trigger is a script that defines the event or condition that initiates the execution of the test case, i.e. the execution of a monitor. A trigger can be set to react to an event (***event-watching***) or to a date (***clock-watching***), and is associated with one or more monitors. Because a trigger initiates the execution of a test case, it is usually not considered as part of the test case itself, but part of the test suite that coordinates the execution of several test cases. A trigger is ***active*** when ready to react to events for which it has been set, and ready to trigger its associated monitors. When a trigger starts a test case, a ***case execution space*** (CES) is allocated, within which the created monitor instance as well as all subsequent dependent instances will execute. The CES defines a single scope of access to events and to other objects referred to by variables. When activated, a trigger is given a ***context*** object, that will be part of the CES of the monitor(s) the trigger will start.

Test Suite: A test suite is a set of test cases, the execution of which is coordinated in some way. This coordination may be represented by a monitor, that will either directly start the monitors that represent individual test cases, or that will instead activate triggers that control these monitors. For

Copyright © OASIS Open 2006. All Rights Reserved.

example, a test suite may serialize the execution of test cases TC1 and TC2 by setting a trigger for TC2 that reacts to the event posted by TC1 at the end of its execution. Or, the test suite may set a trigger that will initiate the concurrent execution of several test cases.
The following figure illustrates the structure of a test suite:

Figure 2. Test Suite.



Event (or Test Event): An event is a timestamped object that is managed by the Event Board. Events are used to coordinate the execution of a test case, and to communicate with external entities. For example an event may serve as a triggering mechanism (in event-driven triggers) for test cases, as a synchronization mechanism (e.g. a test step waiting for an event) or as a proxy for business messages, in which case the mapping between the event representation and the business message is done by an *event adapter*. Some events are ***temporary***, which means they are only visible to monitors from the same test case execution (CES) and are automatically removed from the EB at the end of the CES they are associated with.

Event Board (EB): The event board provides event management functions. Events can be posted to the board, or searched. An event board can be seen as an event log that supports additional management functions. The event board is the main component with which a monitor interacts during its execution, and in many cases the only one.

Event Adapter: An event adapter is a mediator between the external world and the event board. It maps external events such as message sending/receiving, to test events and vice versa. For example, an event adapter will interface with an eBusiness gateway so that it will convert received business messages into a test event and post it on the event board. Conversely, some events

Copyright © OASIS Open 2006. All Rights Reserved.

posted on the event board by a monitor can be automatically converted by the adapter into business messages submitted for sending. An event adapter can also be directly invoked by a monitor. Whether the adapter is designed to react to the posting of an event on the board or is directly invoked by the monitor, is an implementation choice. In both cases, it would convert a test event into an external action.

Eval Adapter: An eval adapter is implementing – or interfacing with an implementation of - a test predicate that requires specific processing of provided inputs that is not supported by the script language. Typically, it supports a validation check, e.g. semantic validation of a business document. An eval adapter is always invoked by a monitor. On invocation, an eval adapter returns an XML infoset summarizing the outcome, that can be evaluated later in the monitor workflow. Document processors fit in this category: Schematron, XSLT, OWL Reasoner...

3.2 Deferred Testing vs Live Testing

The execution of a test case starts at an ***initial date*** (which is actually a dateTime, in the sense of the XML schema datatype). The initial date does not have to be the present date. The execution model is based on a notion of ***time-cursor*** which slides on a virtual time axis, from the initial date and time. The time-cursor represents the virtual present time, for the test case execution. Its scope is the case execution space (two test cases may execute based on different time-cursors, even if executing concurrently in real time).

The difference between live (or real-time) testing and deferred testing depends on the initial date:

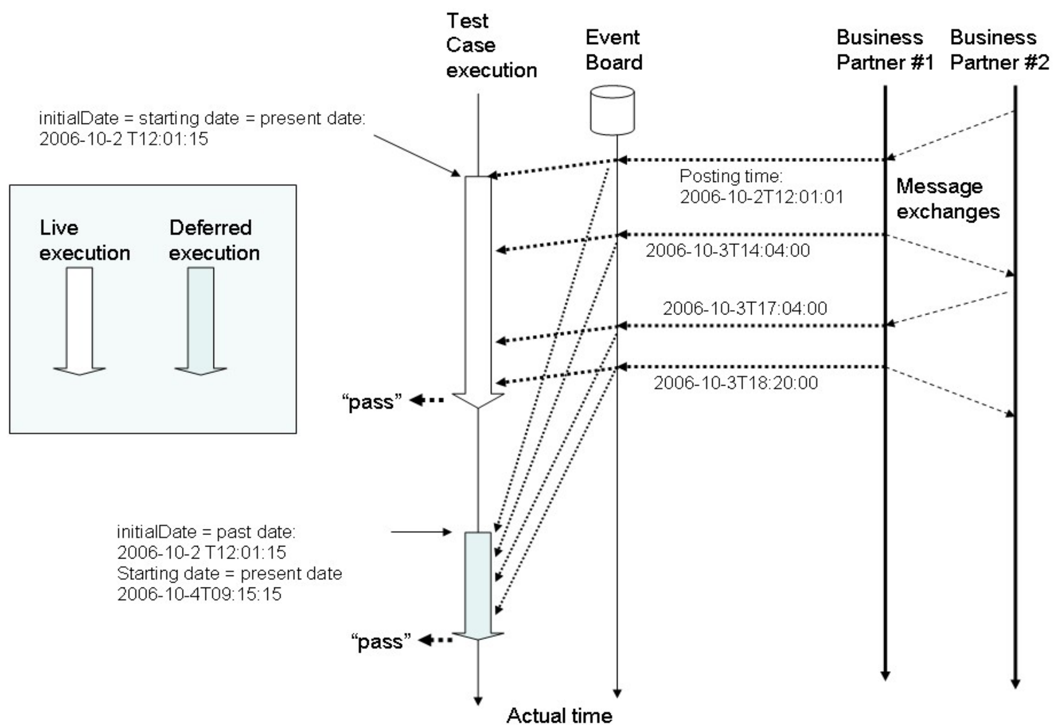
- ***Live testing***: the initial date of the test case is the present date and time. The time-cursor is always the present time.
- ***Deferred testing***: the initial date of the test case is prior to the present date and time. The time-cursor may catch-up with the present time during the execution.

In this document, the time-cursor will always have a value prior or same as the present dateTime, at all times. Cases where the time-cursor can be set to a date ulterior to the present date, are simulation cases out of scope of this version.

The following figure illustrates the same test case executing live and deferred, over the same set of events. Both executions use the same *initialDate*.

Figure 3. Live and Deferred Test Case Executions

Copyright © OASIS Open 2006. All Rights Reserved.



Every time-based operation (*getDateTime* function, *sleep* operation, etc.) is based on the time-cursor and may affect its sliding in turn. For example, a *sleep*(300 sec) in a deferred test case execution will not actually suspend the execution for 300 sec, but may instead slide the time-cursor of 300 sec forward assuming there are no other interferences from concurrent monitors in the same CES. As a consequence, deferred executions are generally shorter than live executions, as illustrated in the previous figure.

Because events on the event board that are time-stamped *after* the time-cursor are not visible to the test case and its operations on the event board, a deferred execution will actually "see" the same events as the original live execution (given the same initialDate).

A test case is said to be ***idempotent*** if any deferred execution of it will produce the same outcome as any previous execution of it, live or deferred, that had the same initial date.

It is not always possible to prove that a test case is idempotent. However, some properties of test cases help ensure this:

A test case is said to be ***passive*** if:

- The test case does not post events on the event board, other than temporary events.
- The test case does not make use of event-adapters.

A test case that satisfies the following properties is idempotent:

- it is passive,
- it either does not invoke eval-adapters, or every invocation of an eval-adapter always returns the same output when given the same input.

Copyright © OASIS Open 2006. All Rights Reserved.

An example of passive test case is one that monitors and validates a business transaction without interfering with its execution.

Note: the activation context of a trigger contains a date (*visibilityDate)* that together with the time-cursor date, defines a window of events in the event board, that are visible to the monitors started by this trigger. If the same trigger is reactivated later with the same context (and therefore same *visibilityDate* and *activationDate)*, then the same set of events will be visible to its monitors, during this deferred testing.

Copyright © OASIS Open 2006. All Rights Reserved.

4.The Event Board

Decisions about the outcome of a test case will often require access to a history of past events. Such an history of events is abstracted here as the “Event Board” (EB). Relying on event analysis for test cases also provides more control on the execution time, which may be deferred.

The intent of this specification is to provide an abstract model of the Event Board and of its operations. The notion of event or “test event” must be understood as an abstraction of any event that may impact the execution of a test case, or result from it, e.g. business messages received or sent, error notifications, outcome of a test case, test operator intervention, etc.

The Event Board also provides operations for managing events. A standard (XML infoset) representation of the invocations of such operations is proposed here, for the purpose of portability of testcase scripts, and leaves the API aspects and its details, to the implementation work. A concrete representation of such operations in terms of language-dependent API calls is out of scope and non-essential to the portability of test scripts: each test engine is supposed to provide its own adapter for converting the infoset representation into an API-specific invocation.

A protocol to capture or publish these events is also out of scope, and is already covered by other specifications (e.g. in form of notification, etc). Although an implementation of the EB will likely act as an event sink, this is not a necessary assumption for this specification: the EB is only assumed (1) to represent a log of events that have occurred, (2) to support an event model (or rather, an event-envelope model) and (3) to provide some form of management of and access to these events.

4.1 The Event Model

Every event posted to the EB is given an envelope, or event-envelope. An event envelope is an object that facilitates the event management. It has the following attributes:

- **iD**: a generated number assigned to an event at the time of its posting to the EB The iD uniquely identifies the event. The iD defines a total order for the events in the board. This order is the same as the time order reflected by the timePost attribute.
- **timePost**: date/time of the event posting.
- **caseId**: the ID of the CES(case execution space) which the monitor instance that generated this event belongs to (in case the event is generated by a monitor).
- **mInstId**: the ID of the monitor instance that generated this event (in case the event is generated by a monitor).
- **evType**: event type, for which there are a few reserved values: “endmonitor”, “endcase”, “endsuite”.
- **temp**: a boolean that is “true” if the event is temporary.
- **evProperties**: a list of name / value pairs open for profiling.

The above attributes are “added” to the original event – or event content - for facilitating the processing of the event in the context of the board. The evProperties items are derived from the content of the original event, and represent an abstraction of it. It may contain references to external objects not managed by the EB. It may be used as a manifest for the actual event content.

The event envelope is represented as an XML infoset of the form:

Copyright © OASIS Open 2006. All Rights Reserved.

[NOTE: RelaxNG notation]

```
<define name=”Event”>
<element  name=”event”  datatypeLibrary="http://www.w3.org/2001/XMLSchema-
datatypes”>
  <attribute name=”id”><data type=”integer”/></attribute>
  <attribute name=”timepost”><data type=”dateTime”/></attribute>
  <attribute name=”evtype”><text/></attribute>
  <attribute name=”temp”><data type=”boolean”/></attribute>
  <optional>
  <attribute name=”caseid”><text/></attribute>
  <attribute name=”minstid”><text/></attribute>
  <element name=”evproperties”>
    <zeroOrMore>
    <element name=”property”>
    <attribute name=”name”/>
    </element>
    </zeroOrMore>
  </element>
  </optional>
  <element name=”content”> <!-- a wrapper for the original event content
-->
  </element>
</element>
</define>
```

the <content> element is a wrapper for any document associated with the event.
Example: An event may be a SOAP message either sent or received. In that case the <content > element contains the SOAP envelope or a subset of it. In case there are attachments (MIME parts) these may remain external to the event envelope representation, and be referred to.

4.2 Event Board Operations

The following operations follow the notational conventions of section 5.
[add Namespaces]

Post (<evtype>? <properties>? <eventContent>?)

Adds an event entry to the event board (EB) – more precisely, adds an event envelope instance. Some attributes of the event envelope are automatically added by the test engine implementation (ID, timePost), others are implicitly passed or set (caseId, minstid). The event type @evtype needs be set for non-predefined events. The event @temp attribute is set to “true” by default. The <evproperties> and <content> elements must be explicitly defined in the post statement, if they must be added to the event envelope. All arguments of the post statement are optional.

```
<define name=”Post”>
<element name=”post”>
<optional>
  <attribute name=”step”><text/></attribute>
  <attribute  name=”evtype”><text/></attribute>
  <element name=”evproperties”>
    <!-- as in event definition -->
  </element>
  </optional>
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
  <element name="content"> <!--  event content -->
  </element>
</optional>
</element>
</define>
```

Find(<tryDuration>? <scope>? <selector> <view>?)

Select one or more non-masked events from the event board within a time window. The selection is based on an XPath expression or an XQuery. The operation may wait for the event(s), acting as a synchronizing control.

To be eligible for selection by this operation, an event of the event board must satisfy the following conditions:

- the event is not masked for the monitor instance executing "find",
- the event posting time (dateTime) is in the window [visibilityDate, time-cursor]
- In case it is a temporary event, it is associated with the same CES as this monitor instance (it has been posted by a monitor instance from the same test case execution).
- The event is within scope (if any, defined by <scope> argument)

Only such events will be considered for selection. If they satisfy the selector element (if any, defined by <selector> argument), then they will be part of the result set for the operation.

The produced selection conforms to the <view> argument if any, that allows for projecting only a subset of event data for each selected event, into the result representation.

The parameters have the following semantics:

tryDuration: intuitively, how long can the Find operation "last" from the (virtual or real) time it starts to execute (called the "effectiveTime"), until the event Board (EB) is in a state where the operation Find returns a positive result (i.e. at least one selected event.) In other words, the Find operation may "query" the event board several times until it returns a positive result, over the time window [effectiveTime, effectiveTime + tryDuration] . The first positive result obtained during these attempts is the final outcome used in subsequent steps of the monitor execution.
If tryDuration is absent: the find operation is executing only once, at effectiveTime (default value for tryDuration is 0).

scope: specifies a set of events of which the event selection must be a subset.
The returned set of selected events will be the intersection of those selected from the EB (satisfying the <selector> element) and those from the scope. The scope is specified either in the same way as the <selector> argument, or as the result of a precious selection.
If not present, the default scope is the entire EB.

NOTE: Because the result of a Find can be fed again as input to another Find, one can decide to break a complex Find condition (say a conjunction) into a sequence of selection steps, which allows for reuse when several selections need be made that share common conditions (allowing for a level of optimization that otherwise would require sophisticated query analysis) . Also, breaking a selection process into steps allows for finer control on the event management (e.g. masking).

selector: specifies a condition that each event to be selected from the EB must satisfy. Its parts are:

selector/@pos: defines the position of an event that will be ultimately selected, among the list of chronologically ordered events that satisfy the condition part. Its value is an Xpath equality

Copyright © OASIS Open 2006. All Rights Reserved.

expression using the function position().

Selector/@starting: defines the earliest date from which events must be selected.

Selector/condition: specifies the condition to be satisfied by selected events from the event board.

Selector/condition/@language: defines the dialect used to express the condition. By default: Xpath2.0. Its set of values includes: “xpath1”, “xpath2”, “xquery1”, “xquery2”.

.

Example:

```
<find step="step1">

<selector pos="position()=1" starting=”initialDate”>

<condition language=”xpath”>[content/soap/Header/msgData/action="PO"]</condition>

</selector>

</find>
```

The selector applies to the sequence of events represented in the EB, that are in scope and also within the time window [visibiityDate, time-cursor] associated with the CES within which the selection occurs.

view: determines how much data from each event record in the EB must be reproduced in the produced selection. It is specified as a set of Xpath expressions. If not present, only the event envelope data is returned, i.e. The event/content element is absent.

```
<define name=”Find”>
<element name=”find”>
<optional>
      <attribute name=”step”><text/></attribute>
      <attribute name=”tryduration”><data type=”duration”/></attribute>
      <element name=”scope”>...</element>
</optional>
<element name=”selector”>
<optional>
      <attribute name=”get”>
      <choice>
            <value>first</value>
            <value>last</value>
            <value>all</value> <!-- default -->
      </choice>
      </attribute>
</optional>
[selection filter: choice of <xpath> or <xquery>]
</element>
<optional>
<element name=”view”>..</element>
</optional>
</element>
</define>
```

The selected set of event is represented as an infoset assigned to the $output local variable.

```
<define name=”Selection”>
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
<element name="selection">
<attribute name="time"><data type="dateTime"/></attribute>
<zeroOrMore>
      <ref name="Event"/>
</zeroOrMore>
</element>
</define>
```

Example:

```
<selection time="date-of-selection">
      <event    evid="1234"    evtype="inmessage"    timepost="2006-05-
15T05:00:02Z">
            <evproperties>...</evproperties>
            <content>...</content>
      </event>
      <event    evid="1237"    evtype="inmessage"    timepost="2006-05-
15T08:35:02Z">
            <evproperties>...</evproperties>
            <content>...</content>
      </event>
      <event    evid="1242"    evtype="inmessage"    timepost="2006-05-
15T11:05:02Z">
            <evproperties>...</evproperties>
            <content>...</content>
      </event>
</selection>
```


Copyright © OASIS Open 2006. All Rights Reserved.

5.Test Case Scripting and Semantics

5.1 Conventions

- Namespace ?
- RelaxNG notation or approximate. The data types used are by default from the xml schema library http://www.w3.org/2001/XMLSchema-datatypes.

5.2 Main Test Case Constructs

Trigger:

Triggers may be activated at any date, past or future. The activation of a trigger means that the trigger is ready to listen to events that it is meant to react to. Reacting to an event typically means triggering (starting) a monitor execution. If the activation date is past, then “listening to events” means scanning past events in the event board that have occurred after the activation date.

There are two types of triggers:

1. Clock-watching trigger: the trigger is scheduled to react at a specific date, or after some elapsed time.
2. Event-watching trigger: the trigger is associated with an event or a class of events, to which it will react, when posted to the event board.

In both cases, the trigger may optionally repeat its triggering, either by waiting again for some time interval (or for same elapsed time from the date it was activated until the first triggering, if interval absent) in case of a clock-watching trigger, or for the next event in case of an event-watching trigger. In both cases, the number of repeats is either bounded by a date, or by a number (or whichever comes first if both bounding criteria are given).

Examples of trigger definitions:

Clock-watching trigger: (informal sample infoset)

```
<trigger label = “myClockTrigger” type=”clock”>
<timer elapse=”[duration]”/>
<repeat until=”[date]” times=”[nbr]” interval=”[duration]”/>
<monitors>...
<!-- list of monitors to be started by the trigger -->
</monitors>
</trigger>
```

Event-watching trigger:

```
<trigger label = “myEventTrigger” type=”Event”>
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
<find>... </find> <!-- some filter on the event board ... -->
<repeat until="[date]" times="[nbr]"/>
<monitors>...</monitors>
</trigger>
```

If <repeat> is present without attributes, the trigger will remain active indefinitely.
If <repeat> is absent, the trigger will not reactivate after it has triggered.

In order to be activated a trigger must be associated with two dates that will be part of its activation context (this context itself is an object that may contain other parameters):

- The a*ctivationDate*, that is the (past, present or future) date and time at which the trigger becomes active. This dateTime value partially determines the initial date of the test cases started by this trigger.
- The *visibilityDate*, that defines a date anterior to activationDate before which the events in the event board are not visible to the test case executions started by this trigger. This date must always be earlier or equal to *activationDate*. By default, it is the same.

Every time a trigger starts a monitor, a new test case execution begins, and a new case execution space (CES) is created. This CES inherits the context object provided by the trigger activation (*actr* operation). The initial monitor started for a test case by a trigger is called the ***root monitor*** *f*or this test case. If the root monitor was started by an event-watching trigger, the output of the find operation used in the trigger is available to the root monitor in the $output variable.

The CES created at each *root start* is defined by:

- an inherited context from trigger activation.
- an object space where all variables and constants are visible to all monitor instances executing in the CES, from the time they are allocated.
- a time-cursor, initiated with the *initial date* of the test case, and used by all monitor instances in this CES.

The time-cursor for a CES is the present date, in case the initial date for the test case was the current date. The initial date is set by the triggering event, which occurs necessarily at a date same or after the activationDate.

For example:

- if activationDate for trigger T1 of type "event" is the past date   2005-01-15T12:00:00, then the trigger will scan events on the Event Board starting from this date, in search for a triggering event. If the triggering event was posted on 2005-01-17T15:22:00, then this time is the initial date (initial value for the time-cursor) given to the root monitor start and to its associated CES. In case the trigger had a <repeat> statement, the next root start will be given an initial date same as the posting time of the next triggering event.
- if trigger T2 of type "clock" has its timer set to the past date 2005-01-20T12:00:00, then this is the initial date given to the root monitor start, becoming its virtual starting time. If T2 has a <repeat> statement with an @interval of 24h, the next root start and its associated CES will have an initial date of 2005-01-21T12:00:00.

<u>Trigger infoset definition</u>:

```
<define name="Trigger">
<element name="trigger">
<attribute name="label"><text/></attribute>
<attribute name="type">
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
<choice>
       <value>clock</value>
       <value>event</value>
</choice>
<optional>
<element name=”repeat”>
<zeroOrMore>
       <attribute name=”until”><data type=”dateTime”></attribute>
       <attribute name=”times”><data type=”integer”></attribute>
       <attribute name=”interval”><data type=”duration”></attribute>
</zeroOrMore>
</element>
</optional>
<choice>
       <element name=”timer”>
              <choice>
              <attribute name=”elapse”><data type=”duration”></attribute>
              <attribute name=”date”><data type=”dateTime”></attribute>
              </choice>
       </element>
       <ref name=”Find”/> <!-- see operation find -->
</choice>
<element name=”monitors”>
       <oneOrMore>
       <element name=”start”>
       ...[see operation “start”]
       </element>
       </oneOrMore>
</element >
</element>
</define>
```

Context

A context is associated with a trigger when the latter is activated. The context contains the optional *activationDate* (default is the present date) and the optional *visibilityDate* date (default is same as the activation date.)
The context is shared by all monitor instances within the case execution space(s) defined by this trigger activation. Additional values may be inserted in the context and associated with a read-only symbol using the assignment element <set>.
Example:

```
<context>
<activationDate>2005-01-20T12:00:00</activationDate>
<visibilityDate>2005-01-15T12:00:00</visibilityDate>
<set cnst=”maxdelay”>1000</set>
...
</context>
```

RelaxNG schema:

```
<define name=”Context”>
<element name=”context”>
<element name=”activationDate”><data type=”dateTime”/></element>
<element name=”visibilityDate”><data type=”dateTime”/></element>
<optional>
       <element name=”set”>...</element>
</optional>
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
</element>
</define>
```

Monitor

A uniquely labeled sequence of test steps, that is activated by a trigger or by a monitor.
The label identifying a monitor is called an “m-label”.

Example:

```
<monitor label=”myname”>
[a sequence of test steps]
</monitor>
```

When an instance of monitor is started, it is automatically given by the test engine an instance Id ($instanceId) unique within the CES.
This instance also inherits:

- the Id of the CES ($caseId) to which this instance belongs.
- The context associated with this execution space.
- All variables and constants declared so far by monitor instances in this execution space.

The test steps in a Monitor always execute in a serialized manner: within a monitor instance, there is always at most one step executing at any time. However, several monitor instances may execute concurrently.
A monitor instance execution is said to be complete either when there is no step succeeding to the last executing step, or when it exits (operation *exit*). A monitor instance may complete while the instances of monitors that it has started are still running.
A test case instance is said to terminate its execution when either all related monitor instances in the same CES completed, or when one of them executes an **exit** statement, interrupting the execution of other incomplete instances in the CES.

Within a CES, all monitor instances have the same visibility on events in the event board, even if they execute concurrently. An event operation (posting, marking) by a monitor step or by an adapter becomes immediately visible to all monitor instances in the same CES.

Across CES, only non-temporary events posted by a monitor in one CES, are immediately visible to another monitor in another CES. Temporary event posting, as well as any event marking done by a monitor within a CES, are not visible to monitors in another CES, and are only effective during the life of the CES.

If a monitor instance activates a trigger that in turn starts another monitor instance, these two monitor instances do not share any context or variables and the new instance is a *root start*, that initiates a new case execution space. Both instances belong to different CES.

Status and Result

The local monitor variable $status is automatically set with the following infoset, at the end of each labelled test step:

```
<define name=”Status”>
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
<element name="status">
<attribute name="step">[test step label]</attribute>
<attribute name="date">[ execution timestamp]</attribute>
<choice>
<value>ok</value>
<value>fail</value>
</choice>
</element>
</define>
```

This infoset is made available at run-time ($status variable) for usability in the test case logic. The value "ok" means that the test step completed as expected. The value "fail" means that an exception condition was met during the execution of the test step. The status is especially useful for reporting on the execution of a step that has delegated to an external adapter.

Example:

```
<status step="stepA" date="...">ok</status>
```

The monitor variable $result is set with the result of the test case. By default it is set to "undetermined". The possible values for this variable are defined by the infoset:

```
<define name="Result">
<element name="result">
<attribute name="step">[test step label]</attribute>
<attribute name="date">[ execution timestamp]</attribute>
<choice>
<value>fail</value>
<value>pass</value>
<value>undetermined</value>
</choice>
</element>
</define>
```

Example:

```
<result step="stepA" date="...">pass</status>
```

An exiting step (exit) is automatically setting the value of this variable to "pass" , "fail", or "undetermined". It can be set (*set* statement) as any other monitor variable. This variable, shared by all monitor instances within a CES, may be used when a test suite is designed to execute its test cases within the same CES, or when a test case is made of several sub-test cases. It helps coordinating the execution of several test cases within a single CES.

Case Trace

Each execution of a test case instance leaves a trace. The trace contains:

- the name of the trigger that initiated the test case
- the name and instance id of every monitor started,
- a sequence of test-steps labels, tracing all executed test-steps that were labeled in every monitor instance that was started, along with the $status value they produced.
- final state of variables used so far in the execution space, including $output, as well as values of all constants used , including those inherited from the context.

Copyright © OASIS Open 2006. All Rights Reserved.

– exit value, in case the monitor execution terminates on an exit operation.

Example:

```
<ctrace caseid=”100” exit=”true”>
<source trigger=”trigA”/>
<teststeps>
<step      mlabel=”monitorError”      name=”start”      minstid=”21435”
initialdate=”2006-05-15T02:00:02Z”/>
<step  mlabel=”monitorA”  name=”start”  minstid=”21437”  initialdate=”2006-
05-15T02:00:03Z/>
<step mlabel=”monitorA” name=”process-PO-message”>ok</step>
<step mlabel=”monitorA” name=”process-Receipt-message”>ok</step>
</teststeps>
<cnst name=”activationDate”>2006-05-15T01:00:02Z</cnst>
<cnst name=”visibilityDate”>2006-05-15T01:00:02Z</cnst>
<cnst name=”maxdelay”>1000</cnst>
<var name="v2”>false</var>

<var name="v1”>123</var>
<var name=”result”>
  <selection time="2006-05-15T11:05:03Z">
       <event     id="1234"     evtype=”inmessage”     timepost=”2006-05-
15T05:00:02Z”><evproperties>...</evproperties><content>...</content></event>
       <event     id="1242"     evtype=”inmessage”     timepost=”2006-05-
15T11:05:02Z”><evproperties>...</evproperties><content>...</content></event>
  </selection>
</var>
<exit time=”2006-05-15T11:05:30Z”>
       <result step=”exitB”>pass</status>
</exit>
</ctrace>
```

When terminating on an exit statement, the ctrace/@exit attribute is set to “true”. This trace is posted as a non-temporary event on the EB. In this event, the <ctrace> element is child of the <content> element. The attribute @evtype is set to “endcase”.

Variables and Constants

Scope

Every variable and constant used in a monitor instance, and that is not a reserved variable (see next section) is dynamically declared and allocated in the CES (case execution space) the first time a <set> statement concerning this object is executed. This object is then immediately accessible - and modifiable if a variable – to all concurrent and subsequent monitor instances executing in this CES, even after the monitor instance that declared it is complete. This means:

- If Monitor instance m1 defines variable $aa, then starts (concurrently or not) Monitor instance m2 which sets $aa to some value, then any subsequent step in m1 after the change done by m2, will see this modification.
- If Monitor instance m1 starts (concurrently or not) Monitor instance m2 which defines and sets a variable $aa to some value, then any subsequent step in m1 referring to $aa after it has been set by m2 will see this value and be able to modify it.

There are however two exceptions: the reserved $status variable and the $output variable are always only local to their current monitor instance, and always refer to different status objects and

Copyright © OASIS Open 2006. All Rights Reserved.

result objects when used by different monitor instances, concurrently or not. Consequently they are not available anymore when a monitor instance completes unless assigned to another variable. In a monitor instance that exits, these local variables will also be reported in the <ctrace>.

Usage

Variable and constant identifiers are of the form $abc, where “abc” is an alphanumeric string. Variables are not typed. The same variable may either contain an XML infoset or an atomic value (which may or may not be converted to the expected type at the time it is consumed by a function or processor).

The assignment of the value “myvalue” to the variable $myvar is expressed as a “set statement” of the form (the prefix “$” is omitted when the name of the variable itself is used as an XML attribute value):

```
<set var=”myvar”>myvalue</var>
```

Such a statement also implicitly declares a variable. The assignment of the variable $myvar2 to the variable $myvar1 (copying its value) is expressed as:

```
<set var=”myvar1” source=”var”>myvar2</var>
```

The assignment of the value (or infoset) identified by the Xpath: /event/content/PurchaseOrder/OrderRef over the document referred by the variable $output, is expressed:

```
<set var=”myvar” source=”xpath”>
$output/event/content/PurchaseOrder/OrderRef</var>
```

When using a constant instead of a variable, the statement is:

```
<set cnst=”myconst”>myvalue</var>
```

The script language does not support any more complex assignment or expression: the evaluation of a complex expression must be delegated to an eval-adapter.

Template variables

A variable notation ($abc) may be used in a document, e.g. an XML document, at any place. Such a parameterized document is called a template. Such a parameterization may be defined independently from the scripting of monitors that will use it. In order to avoid conflicts between template variables and monitor variables, the substitution of template variables is only done within a <template> statement used as a child of the <set> statement:

```
<set var=”mynewdoc” source=”template”>
<template [href=”...”?] [document=”purchaseorder”]>
       <set var=”customer”>Mary</set>
       <set var=”product” source=”var”>$pref</set>
</template>
</set>
```

In this example, every occurrence of the string “$customer” in the document assigned to the monitor variable $purchaseorder , will be substituted with “Mary”. Every occurrence of “$product” will be substituted with the content of the monitor variable $pref. The <template> element above is a

Copyright © OASIS Open 2006. All Rights Reserved.

substitute for the end-result of these substitutions.

Other attributes of the <set> statement are:

- set/ @source: with value= “template”, indicates that the value of the <set> element is a <template> element that needs to be instantiated first before being assigned. The variable in set/ @var ($mynewdoc in above example)is set to the result of the template substitution. Other values: “var” (if the value of the set element is a variable name), “xpath” (default) if the value of the set element is an xpath expression the result of which needs be assigned.
- set/ @copy: this optional attribute contains a monitor variable name, to which the content of the <set> element will also be assigned so that it is also .accessible in subsequent steps of the monitor. In other words, <set> with a @copy attributes combines two assignments.

Other forms of template usage and substitution may be supported via appropriate eval-adapters.

Reserved variables and constants

The following variables and constants are reserved and have special meaning:

$caseid

This constant identifies the CES (case execution space) where the current monitor instance belongs. Any event posted by this monitor will have an attribute of same name set to this value. The caseid constant is set at the time a trigger is actually starting a monitor and will be inherited by all monitor instances subsequently started by this initial monitor instance. If a trigger starts several monitors concurrently at once, each will have a different case execution space and therefore different caseid. If a trigger repeats periodically the starting of the same monitor, each instance will have a different execution space and $caseid value.

$instid

This constant identifies the current monitor instance.

$activationDate

Constant that contains the activation date of the originating trigger, by default the present time when the trigger activation statement was executed. Part of the context of execution.

$visibilityDate

Constant that contains the date before which events of the event board that are not visible to this execution space. By default, same as $activtionDate. However, this date can be set to an earlier date – e.g. in case a monitor needs to be able to correlate events that occurred before the activation date. Part of the context of execution.

$result

This variable is set with the final result of a test case executio (see “status and result” subsection.)

$status

This local variable is implicitly updated by every test step (see “status and result” subsection.)

Copyright © OASIS Open 2006. All Rights Reserved.

$output

This local variable is implicitly updated by any test step that produces results, such as **find**, **call** (for an eval-adapter). The infoset may vary for a **call** step. For a **find** step, it is a <selection> infoset as in section 4.2:

5.3 Scripting of Test Steps

Monitor step (or m-step)

A monitor step is an atomic unit of monitor execution.

Every step is setting the local $status variable to either “ok” or “fail” (see format in section on variables), at the end of its execution.
Every step may optionally be labelled (uniquely identified within a Monitor).
Some steps will set the local $output variable with a value or an infoset, in addition to setting $status.

A step may do one of the following 11 operations (except for *cad*, which can embed another of these operations):

Event operations:

- **post:** generate an event and post it on the Event Board.
- **find:** selects event(s) from the Event Board.
- **mask:** mask or unmask some past events to a monitor instance, and to all monitor instances executing in the same space.

Monitor flow control:

- **start:** starts a new instance of another monitor or of current monitor, either synchronously to the current instance, or asynchronously.
- **set:** assigns a value or an infoset to a variable or to a constant. Automatically declares and allocates the variable or constant if not previously used. May be used just once in the scope of a CES for constants.
- **sleep :** the monitor execution is suspended either until a specified date or for a specified duration. The duration may be virtual in case of deferred execution.
- **cad:** check-and-do operation that verifies a predicate on the result of one or more previous test steps and does a single operation in case the verification is positive; e.g., exit the test case. It allows for optional branching to another labeled step after the "do" part.
- **jump**: pursue the execution thread at another (labeled) step in the monitor.

External resources:

- **call:** either an event-adapter or an evaluation-adapter is invoked. The invocation of an

Copyright © OASIS Open 2006. All Rights Reserved.

event-adapter will result in a message generation/sending from an event or from input argument(s). Invoking an evaluation-adapter will produce an XML infoset. One or more input infosets are passed as argument (s). An external event (e.g. message sending) will be generated in case of an event adapter.

Test case control:

- **actr**: dynamically activate a trigger. This will schedule the execution of its monitors - either based on events that may not have yet occurred, or based on a future date. The monitor execution resumes immediately after activating the trigger.
- **exit :** terminates the current test case along with all currently executing monitor instances in the case execution space. Produces a result of either pass, fail or undetermined. A trace of the test case execution and status are posted as events to the event board

post (see Event Board Operation section)

find (see Event Board Operation section)

mask

Mask (<EventSelection >?<value>?)

The operation either masks or unmasks the events in the board that match the <eventSelection>. The default value for mask is “true” (the events will be masked after the operation)

```
<element name=”mask”>
<optional>
<attribute name=”step”><text/></attribute>
<attribute name=”value”><data type=”boolean”/></attribute>
</optional>
<ref name=”Selection”>
</element>
```

Events are only masked relatively to a case execution space, i.e. for all monitors instances that have same $caseid as the monitor instance doing the masking. Such events cannot be selected (find operation) anymore by these instances or subsequent instances in the same CES. This feature provides a way to add some statefullness to the event board, relative to the consumption of events by monitors. For example, in many cases (though not always) an event should not be consumed twice.

Events are never masked for triggers: when an event-watching trigger activates, even if activated from a monitor, it will ignore the masking set by this monitor or the monitors of same case execution space. Indeed, this trigger will set a new case execution space that will ignore all masking.

start:

The script for this operation is:

```
<element name=”start”>
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
<optional>
      <attribute name="step"><text/></attribute>
      <attribute name="instancevar"><text/></attribute>
      <attribute name="casevar"><text/></attribute>
      <attribute name="sync"><data type="boolean"/></attribute>
</optional>
<attribute name="mlabel"><text/></attribute>
</element>
```

Example 1:

```
<monitor label="monitor0">
...
<start step="step9" instancevar="myinst" sync="true" mlabel="monitor1"/>
...
</monitor>
```

In this example, an instance of “monitor1” is started by an instance of “monitor0”.
The attribute @instancevar contains the name of the variable ($myinst in the example) to which the instId of the new “monitor1” instance will automatically be assigned.
The boolean attribute @sync specifies whether the monitor instance is started synchronously (“true”), meaning the current flow of “monitor0” will resume only when the execution of monitor1 completes. If @sync=”false” (default), the next test step of monitor0 executes immediately, concurrently to monitor1.

Example 2:

```
<monitor label="monitor0">
...
<trigger label="trigger0">
...
<monitors>
<start casevar="mycase" mlabel="monitor1"/>
</monitors>
</trigger>
...
</monitor>
```

In this example, an instance of “monitor1” is started by a trigger, itself activated from monitor0.
The attribute @casevar contains the name of the variable ($mycase in the example) to which the CES id associated with the start of the “monitor1” instance will automatically be assigned. Monitor0 may then use $mycase later, for example to track the ending event of the test case started with monitor1.

set (see section 5.2)

sleep:

Suspends the execution of the current monitor instance for some duration, or until some date. More precisely, slides the time-cursor of the specified duration (from the effective dateTime of the operation) or until the specified dateTime.

```
<element name="sleep">
<optional>
      <attribute name="step"><text/></attribute>
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
</optional>
<choice>
      <attribute name="until"><data type="dateTime"></attribute>
      <attribute name="duration"><data type="integer"></attribute>
</choice>
</element>
```

Example:

```
<sleep step="step3" duration="300"/>
```

cad (check-and-do)

The operation allows for branching to a different step of the monitor based on the result of a logical expression, that may include any variable or infoset within the CES scope. The expression in the condition element may be an XPath2.0 logical expression (“xplog”).

```
<element name="cad">
<optional>
      <attribute name="step"><text/></attribute>
</optional>
<element name="condition">

            <!-- reuse some standard XML expression syntax? RuleML, PRR?
-->
            <attribute name="language"> ...</attribute  >


</element>
<element name="do">
      <optional>
            <attribute name="thenjump"><text/></attribute>
            <choice>
            <ref name="[any monitor operation]"/>
            </choice>
      </optional>
</element>
</element>
```

If the condition is satisfied, the single action specified in the element <do> is executed, then optionally a jump is performed to another step specified in @thenjump. Else, the <cad> step is complete, and the next step in the monitor instance is executed. The action child of the <do> element is any of the monitor steps described in this section, except for *cad*, which cannot be nested in another *cad*.

Example:

```
<cad step="step3">
<condition
      language="xpath1">($status = "ok") and ($bustx = 'purchaseorder'
</condition>
<do thenjump="step6">
      <post>
            <evtype>POstatus</evtype>
            <evproperties>
            <property name="purchaseorder">$poref</property>
            </evproperties>
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
        </post>
</do>
</cad>
```

Note: the controls operators are intentionally primitive but sufficient. The script language does not intend to replicate programming languages: no nested structure, conditional or switch statement, or loop statements. The structure is deliberately flat:: there is no other grouping construct than the monitor. Steps cannot be nested into steps (when a step starts a monitor, this step is complete). The action in the <do> statement (*post* in the example) should not use a @step attribute.
In this flat execution model the only context to a step execution is the CES state. This simplicity and the primitiveness of operators for controlling the execution flow may produce inelegant scripts in some cases, but the logic of these is easier to trace, verify and validate, w/r to events and timing, which is essential for a test case.

If several steps need be done for a positive outcome of the <condition> element of the *cad* operation, then they should either be grouped in a new monitor, or should be defined outside the <cad> operator starting at the @thenjump step (in which case, the <do> element is empty).

jump

The operation causes the monitor execution to continue at the specified labeled step within the monitor. The label of the step is given in the attribute @tostep.

```
<element name=”jump”>
<optional>
      <attribute name=”step”><text/></attribute>
</optional>
<attribute name=”tostep”><text/></attribute>
</element>
```

actr

Activates a trigger, making the trigger receptive to future triggering events (in case of live testing) or to past triggering events (in case of deferred testing). The script for this operation is:

```
<element name=”actr”>
<optional>
      <attribute name=”step”><text/></attribute>
</optional>
<ref name=”Context”/>
<ref name=”Trigger”/>
</element>
```

The context element specific to this activation is passed to the trigger and will in turn be inherited by all the monitors instances started in the same case execution space.

Example:

```
<actr step=”step4”>
<context>
      <activationdate>2006-08-25T13:30:00Z</activationdate>
      <set cnst=”maxTimeToPerform”>3600</set>
      <set cnst=”maxAcknowledgementTime”>600</set>
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
</context>
<trigger label="T1" type="Event">
        <find get="first">
        <selector><condition language="xpath1">
              event[@evtype="startMyTestSuite"]
        </condition></selector>
        </find>
        <monitors>
              <start instancevar="instm1">monitor1</start>
        </monitors>
</trigger>
</actr>
```

The @activationdate attribute sets the date at which the trigger is first activated, which may be a past date in case of deferred activation. If absent, the current date is the activation date.

The name of the trigger could also be provided instead of the full trigger definition:

```
<trigger label="T1" />
```

exit:

Terminates the execution of a test case and of all related monitor instances within the same case execution space. The value attribute of the <exit> element will automatically set the value of the $result variable. It may be: pass, fail, undetermined.

```
<exit step="step7" value="fail"/>
```

A trace event with @evtype= “endcase” containing the trace of the test case (case trace) is posted to the EB (see ctrace section).

call (Event Adapter)

The operation interfaces with an external processor. A set of numbered parameters can be passed, each one of them containing a value (atomic value, infoset, or document template instantiation with the <template> statement) or a variables referring to such a value.   The number and expected content of arguments vary with each adapter.

Example:

```
<call step="step1" adapter="sendsoap" type="event">
<param arg="1">
<template document="purchaseorder">
<set var="POref">P1234</set> <!-- set Poref in payload -->
<set var="convId">100</set>
<set var="mesgId">111444777</set>
</template>
</param>
<param arg="2">$headerinfo</param>
<param arg="3">$destinationURL</param>
</call>
```

The general semantics is of a function: In all cases, the outcome of the call will set the $status

Copyright © OASIS Open 2006. All Rights Reserved.

variable. in case a more complex result than a status is expected (in addition to status) this result is assigned to the $output variable, The role of the adapter is to interpret the results of the external processor, and mediate with the monitor execution by setting these monitor variables appropriately.

Copyright © OASIS Open 2006. All Rights Reserved.

6.Use Cases

(namespaces are omitted in the scripts)

6.1 Use Case 1: Business transaction monitoring

In this use case, instances of a request-response business transaction are monitored for their compliance with the transaction requirements. The pattern observed must be the following:

```
Step 1: Buyer Sends a P.O. to Supplier.
Step 2: Buyer Receives a P.O. Confirmation or Rejection, correlated with
the P.O.

Constraints:
–  total time for 1-2 is bounded by 300 sec.
–  The correlation is based on a PO reference # that is in the payload.

Success conditions:
–  within 300 sec, receive either a single P.O. Confirmation or Rejection
   (not two).
–  Both messages should share the same ConversationID message header
   attribute.

Failure cases for the transaction:

–  an error message is received, that correlates with the initial message
   based on messageId (failure "A")
–  there was more than 1 response message within 300 sec. (failure "B")
–  no response received within 300 sec. (failure "C")
```

<u>Option 1:</u>

Scripting for a test case where the Test Driver (executing the test case) is simulating the Buyer side in order to test the Supplier side for conformance.

```
<!-- ========== step 1: send the purchase order message -->
<call step="step1" adapter="sendsoap" type="event">
<param arg="1">
<template document="purchaseorder">
<set var="POref" copy="mPOref">P1234</set> <!-- set Poref in payload -->
<set var="convId" copy="mconvId">100</set>
<set var="mesgId" copy="mmesgId">111444777</set>
</template>
</param>
</call>
<!-- ========== : sleep 300 sec -->
<sleep duration="300"/>
<!-- ========== step 2: get related error message(s) if any -->
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
<find step="step2">
<selector>
<condition
language="xpath1">content/soap/Header/messageInfo[type="Error"][RefToMess
ageId=$mmesgId]</condition>
</selector>
</find>
<!-- ========== step 3: check exit failure case A (exist an error
message?) -->
<cad step="step3">
<condition language="xpath1">$output/event</condition>
<do>
<exit value="fail"/>
</do>
</cad>
<!-- ========== step 4: get related response message(s) if any -->
<find step="step4">
<selector>
<condition
language="xpath1">content/soap/Header/messageData[POReference=$mPOref][ac
tion="confirm" or action="reject"]</condition>
</selector>
</find>
<!-- ========== step 5: check success case  (1 response) -->
<cad step="step5">
<condition language="xpath1">
       count($output/event) = 1
</condition>
<do>
<exit value="pass"/>
</do>
</cad>
<!-- ========== step 6: check exit failure case B (too many responses) --
>
<cad step="step6">
<condition language="xpath1">
       count($output/event) &gt; 1
</condition>
<do>
<exit value="fail"/>
</do>
</cad>
<!-- ========== step 6:  exit failure case C (no responses) -->
<exit step="step7" value="fail"/>
</monitor>
```

Note: every failure case is annotated in the monitor trace with a different label (step name).

Option 2:

Scripting for a test case intended to be used for monitoring a deployed application. The test case is converted into a *passive* test case, which does not control the generation of messages: all messages from Buyer or Supplier are supposed to be posted to the Event Board by other agents.

```
<monitor label="mBuyerSupplier">
<!-- === step 1:  the  PO message in the event board to be used
has already been selected by the trigger (in $output)  -->

<!-- set some variables to elements of selected PO message event,
for later use -->
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
<set var="POref"
source=”xpath1”>$output/event/content/soap/Body/PO/POReference</s
et>
<set var="mId"
source=”xpath1”>$output/event/content/soap/Header/msgData/MsgId</
set>
<!-- === : sleep 300 seconds -->
<sleep duration="300"/>
<!-- === step 2: get related error message(s) if any -->
<find step="step2"><selector starting=”initialDate”>
<condition language=”xpath1”>
content/soap/Header/msgInfo[type="Error"][RefToMsgId=$mId]
</condition>
</selector></find>
<!-- === step 3: check exit failure case F1 (one or more error
messages) -->
<cad step="step3">
<condition language=”xpath1”>$output/event
</condition>
<do><exit value="fail"/></do>
</cad>
<!-- === step 4: get related response message(s) if any -->
<find step="step4"><selector starting=”initialDate”>
<condition
language=”xpath1”>content/soap/Header/msgData[POReference=$POref]
[action = "confirm" or action =
"reject"]</condition></selector></find>
<!-- === step 5: check success case (exactly 1 response) -->
<cad step="step5">
<condition language=”xpath1”>count($output/event) = 1</condition>
<do><exit value="pass"/></do>
</cad>
<!-- === step 6: check exit failure case F2 (too many responses)
-->
<cad step="step6">
<condition language=”xpath1”>count($output/event) &gt; 1
</condition>
<do><exit value="fail"/></do>
</cad>
<!-- === step 7: exit failure case F3 (no responses) -->
<exit step="step7" value="fail"/>
</monitor>
```

The test case execution will be controlled by the following trigger. This trigger is itself activated (operation <actr>) at a specific date/time (activation date) within a monitor named "init" that can be started via a command line interface. The activation date in the <context> element defines the earliest date at which events may be selected by the trigger in order to start a test case. The <repeat> element will cause the trigger to iterate on the next PO event selected, and so on until the last PO of July 2006. In each iteration, a new instance of the mBuyerSupplier monitor is started with a different initialDate value set by the time of the triggering event. This demonstrates the continuous monitoring functionality of the eTSL.

```
<monitor label="init"><!-- === Next, just activate the trigger -->
<actr>
<context>
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
<activationDate>2006-07-01T00:00:01Z</activationDate>
</context>
<trigger name="JulyBatch" type="Event">
<find step="step1" tryduration="100000">
      <selector pos="position()=1">
      <condition
language="xpath">content/soap/Header/msgData/[action="PO"]
</condition>
      </selector>
</find>
  <repeat until="2006-07-31T23:59:59Z"/>
      <monitors>
      <start casevar="mycesid" mlabel="mBuyerSupplier"/>
      </monitors>
</trigger>
</actr>
</monitor>
```

NOTE: the use of tryduration="100000" allows the monitor for waiting for the first PO event to be posted (without that atttibute, the monitor would require the PO event to be already in the EB for the find to be successful.)
The test case could have been more efficiently coded: it does not handle cases where getting the initial purchase order in the EB may take a long time (the above test case will only wait 100000 sec), as it must be started before the PO is logged. Also, the above script does not automatically spawn one monitor instance for each PO. By catching the initial PO event with a trigger instead of Step 1 in the monitor, the monitorBuyerSupplier monitor can then be triggered for every one of such events. This is done in Option 3.

Option 3:

In this option, all PO transactions will be monitored, starting from <activationDate>. Note that this date could be past: in that case the monitoring would operate on past PO events on the Event Board (deferred testing). Because the trigger repeats indefinitely, the monitoring will "catch-up" with current time and will continue "live". The following initialization monitor will activate the trigger with the desired activation date:

```
<monitor label="init">
<!-- ========== just activate the trigger  -->
<actr>
<context>
<activationDate>2006-07-15T14:00:00Z</activationDate>
</context>
<trigger name="T1" type="Event">
<find step="step1">
  <selector pos="position() = 1">
                                                              <condition
language="xpath1">content/soap/Header/messageData[action="purchaseOrder"]
</condition>
  </selector>
</find>
<repeat/>
<monitors>
<start casevar="mycesid" mlabel="monitorBuyerSupplier"/>
</monitors>
</trigger>
</actr>
</monitor>
```


Copyright © OASIS Open 2006. All Rights Reserved.

The test case monitor `monitorBuyerSupplier` is similar as for Option 2, except for the missing @tryDuration attribute: the event consumed by the trigger that started this monitor instance, is again available to the monitor instance and therefore there is no need to “wait” for this PO. When the same trigger will start the next monitor instance (when the next PO is posted), the time-cursor for the next instance will be set to the posting time for the new PO event, and the new monitor instance will not “see” the previous PO event.

```
<monitor label=”monitorBuyerSupplier”>
<!-- ========== step 1: get the purchase order message -->
<find step=”step1”>
<selector pos=”position() = 1”>
<condition
language=”xpath1”>content/soap/Header/messageData[action=”purchaseOrder”]
</condition>
</selector>
</find>
<!-- ========== : set some variables to some result values, for later use
-->
...
</monitor>
```

6.2 Use Case 2: Scripting a Test Suite

There is no special construct for Test Suite. A test suite is normally represented as a set of triggers which, when activated will start in due time the monitors representing the test cases within the suite. This will be the first way to script a test suite (Option 1 below).

Consider the following test suite named “myTestSuite“. It contains three test cases, TC1, TC2, TC3 controlled by 3 monitors:
TC1 is scripted by the previous monitor in Use Case 1: verifySupplierSide
TC2 is scripted by a monitor not described here, of name: monitor2.
TC3 is scripted by a monitor not described here, of name: monitor3.

- TC1 and TC2 are started as soon as an event of type “startMyTestSuite “ appears on the event board.
- TC3 is started when TC1 terminates (at the posting of its termination event)
- At the end of the execution of myTestSuite, when termination events have been posted for TC2 and TC3, a test suite termination event of type “endsuite” is posted

<u>Option 1:</u>

This set of triggers can itself be activated within a single Monitor script that would represent the test suite.

The following test suite named “myTestSuite“ will control the execution of its test cases with the flow illustrated in the following figure:

The test suite execution will be triggered by the posting of an event of type “startMyTestSuite” to the event board (the origin of this event is not significant here). The corresponding test suite script is:

```
<monitor label=”myTestSuite”>
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
<!-- ========== : activate trigger T1 -->
<actr>
<context></context>
<trigger name="T1" type="Event">
<find pos="position()=1">
<selector>
<condition
language="xpath1">event[@evtype="startMyTestSuite"]</condition>
</selector>
</find>
<monitors>
<start casevar="cesid1">verifySupplierSide</start>
<start casevar="cesid3">monitor3</start>
</monitors>
</trigger>
</actr>
<!-- ========== : activate trigger T2 -->
<actr>
<context></context>
<trigger name="T2" type="Event">
<find pos="position()=1">
<selector>
<condition          language="xpath1">event[@evtype="endcase"          and
@caseid=$cesid1]</condition>
</selector>
</find>
<monitors>
<start casevar="cesid2">monitor2</start>
</monitors>
</trigger>
</actr>
</monitor>
```

NOTE: these triggers are not executed as steps (like test case steps). They are all activated as soon as this monitor is executed. The monitor “myTestSuite” as written above, may complete even before the test suite execution actually started (i.e. Before the three triggered monitors are started.) as the completion of trigger activation (actr) does not need to wait for the associated monitors to complete or even start. Activating a trigger is equivalent to scheduling some future monitor activity. Once the trigger is active, it remains so until the triggering is complete i.e. until the associated monitors start executing.

Option 2:

A test suite may also be represented as a single monitor that directly coordinates a set of monitors (the test cases) in a serialized or concurrent manner, without making use of triggers. In that case a single trigger is needed to start the entire test suite. With this second option, the scripting of the test suite will be shorter, but the test cases will be more dependent from each other: they will share the same case execution space, i.e. same variables, and same event masks. In fact, from eTSL viewpoint, they all execute in a same test case. Consequently, if one monitor exits, the entire suite is terminated, so some precaution has to be taken for the scripting e.g. avoid the use of exit, and instead explicitly post the result event for each test case.

The following test suite is named “myOtherTestSuite“ and is controlled by a single monitor of same name. It contains three test cases, TC1, TC2, TC3 controlled by monitors of similar names as in option 1 not described here.

Copyright © OASIS Open 2006. All Rights Reserved.

- TC1 and TC2 are started as soon as an event of type “startMyTestSuite “ appears on the event board.
- TC3 is started when TC1 terminates (at the posting of its termination event)
- At the end of the execution of myTestSuite, when termination events have been posted for TC2 and TC3, a test suite termination event of type “endsuite” is posted (not done in Option 1)..

The corresponding test suite script is:

```
<monitor label=”myOtherTestSuite”>

<!-- ========== : wait for the starting event -->
<find step=”step1” get=”first”>
<selector>
<condition
language=”xpath1”>event[@evtype=”startMyTestSuite”]</condition>
</selector>
</find>
<!-- ========== : activate TC1 and TC3 concurrently -->
<start step=”step2” instancevar=”instm1”>verifySupplierSide</start>
<start step=”step3” instancevar=”instm3”>monitor3</start>
<!-- ========== : wait for TC1 to terminate -->
<find step=”step4”  pos=”position() = 1”>
<selector>
<condition        language=”xpath1”>event[@evtype=”endmonitor”        and
@minstid=$instm1]</condition>
</selector>
</find>
<!-- ========== : activate  TC2 after TC1 terminates -->
<start step=”step5” instancevar=”instm2”>monitor2</start>
<!-- ========== : wait for TC2 and TC3 to terminate (order unimportant)
-->
<find step=”step6” pos=”position() = 1”>
<selector>
<condition        language=”xpath1”>event[@evtype=”endmonitor”        and
@minstid=$instm2]</condition>
</selector>
</find>
<find step=”step7” pos=”position() = 1”>
<selector>
<condition        language=”xpath1”>event[@evtype=”endmonitor”        and
@minstid=$instm3]</condition>
</selector>
</find>
<!-- ========== : post  test suite termination event -->
<post step=”step8”>
<evtype>endsuite</evtype>
<evproperties>
<property name=”suite”>myOtherTestSuite</property>
</evproperties>
<!-- other evt attributes automatically set -->
</post>
</monitor>
```


Copyright © OASIS Open 2006. All Rights Reserved.

6.3 Use Case 3: Combined validation of Business Document and Transaction

In this use case, instances of a request-response business transaction are monitored for their compliance with the transaction requirements, along with the semantic validation (not just schema validation) of the business document.

6.4 Use Case 4: Selective Reliability for Some Business Transactions

In this use case, some level of reliability and security is expected for a subset of Purchase Orders, depending on the type of product involved. Some business transaction properties (response time) also depend on it.

Considering the situation when a company has Web services using diverse ways of declaring its security mechanism (e.g., WS-SecurityPolicy, some of them may not have a declaration at all), monitoring for compliance at the message level becomes necessary. This section illustrates a simple usage of eTSL for monitoring all transactions about “Money” in case these are required to use an appropriate security mechanism, in this case *Aes128* encryption or above.

The following monitor does this verification:

```
<monitor label="CheckSecurityMoneyRelatedTransaction">
<!-- === Set the target element/attribute conforming to XML Encryption,
that indicates the algorithm used. -->
<set
var="encAlgo">$output/event/content/soapenv/Body/e:EncryptedData/e:Encryp
tionMethod/@Algorithm</set>
<!-- === Check that the element/attribute specify the encryption
algorithm Aes128 or higher. -->
<cad step="step1">
       <condition language=”xpath1”>[not(contains($encAlgo,"Aes128") or
contains($encAlgo,"Aes192") or contains($encAlgo,"Aes256") )]</condition>
       <do><exit value="fail" label=”fail-1”/></do>
</cad>
<!--=== Exit on success -->
<exit step="step2" value="pass" label=”pass-1"/>
</monitor>
```

The corresponding trigger script is:

```
<monitor label="init"><!-- === activate the trigger that iterates on
Money transfer messages -->
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
<actr>
<context><activationDate>2007-08-01T00:00:01Z</activationDate></context>
<trigger name="AugustMoneyTransfersStatus" watch="Event">
<!--=== Get the next message where the wsa:Action header element has the
term “Money” -->
<find step="step1"><selector pos="position()=1">
 <condition language=”xpath”>[contains(content/soapenv/Header/wsa:Action,
”Money")]</condition>
</selector>
</find>
<repeat until="2007-08-31T23:59:59Z"/>
      <monitors><start                                   casevar="mycesid"
mlabel="CheckSecurityMoneyRelatedTransaction"/> </monitors>
</trigger>
</actr>
</monitor>
```

6.5 Additional Use Cases

Use Case #5: conditional branching

Test Case: (test driver plays "buyer")

- buyer sends M1
- received M2 (e.g. an approval, or rejection)

---> if M2 is "approval", we will expect a sequence of:

- receive M3 (e.g. a quotation)
- send M4 (e.g. approval of quote and payment info)
- receive M5 (final delivery details).

---> if M2 is "rejection", we will expect a sequence of:

- receive M6 (proposed alternative)

We need to verify all received messages, as the test case would fail if they do not comply.
Any error message received, would also fail the test case (see use case #2)

Use Case #6: usages of the EB

Use cases in "getting [test] events" within a Test Case. Events are here messages
sent by the other party.

1. test case keeps going if a response message correlating (condition R) with previous message
is received within time window T. The received message must then satisfy some condition C.
-> Find("tryduration" with max duration T - the filter does only R, not C)
If Find succeeds (returns positive result), test case proceeds with checking C.

2. test case fails if an Error message (condition E) related to previous

Copyright © OASIS Open 2006. All Rights Reserved.

```
message (condition R),
is received within time window T.
-> Find("lasting" with max duration T - the filter does checks E and R)
If Find returns positive result, test case fails.

2. test case succeeds only if exactly 3 message of a particular profile
(condition P) e.g. resulting
from resending mechanism in reliable messaging have been received, within
time window.
-> Find("scheduled" at the timeout of window (now+T), "instantaneous"
execution, checks P)
If Find returns positive result (exactly 3 messages that satisfy P), test
case succeeds.
```

When the adapter is exclusively controlled by the event board (not invoked by the monitor), the execution model for an event adapter is comparable to a looping monitor:

- (a) find(event of the type consumed by the adapter)
- (b) invoke adapter (convert event and send message)
- (c) branch to (a)

Copyright © OASIS Open 2006. All Rights Reserved.

7.Operational Requirements

Requirements from NIST for the Test Event Log (3): Recovery of test case execution.

Requirements from NIST for the Test Event Log (4): Internal state for test case execution.

Copyright © OASIS Open 2006. All Rights Reserved.

8. References

[1] W3C, XSL Transformations (XSLT) 2.0, November 2005.

[2] Hohpe, G. and Woolf, B.. "Enterprise Integration Patterns" Addison-Wesley signature series, August 2006.

[3] UNECE, UN/CEFACT's Modeling Methodology, UMM Meta-Model – Foundation Module V1.0, October 2006.

[4] Java XML Unit (JXUnit), http://jxunit.sourceforge.net.

[5] JUnit, Java for Unit Test, http://junit.sourceforge.net.

[6] ETSI, ES 201 873-x, V3.1.1, Testing and Test Control Notation 3 Standard Suite, 2005-2006.

[7] OASIS, Business Process Specification Schema 1.0.1, May 2001 and ebBP, v2.0.4, October 2006.

[8] Web Services Choreography Description Language (WSCDL), Version 1.0, (candidate recommendation) November 2005.

[9] OASIS, Web Services Business Process Execution Language 2.0 (committee draft in review phase), August 2006.

[10] OASIS, ebXML Implementation, Interoperability and Conformance Test Framework 1.1, October 2004.

[11] W3C, XML Path Language 2.0, June 2006.

[12] W3C, XML Query Language 1.0, June 2006.

[13] OASIS, ebXML Messaging Service Specification 2.0, April 2002.

[14] OASIS Simple Object Access Protocol 1.2, June 2003.

[15] XPDL: XML Process Definition Language) (Workflow Management Coalition) Document Number WFMC-TC-1025: Version 1.14 October 3, 2005.

[16] RosettaNet Ready™, RossettaNet Self-Test Kit, December 2002.

[17] Web Service Interoperability Consortium, Web Service Profiles and Testing Tools, http://www.ws-i.org/deliverables/workinggroup.aspx?wg=testingtools, June 2003.

Copyright © OASIS Open 2006. All Rights Reserved.

[18] IEEE, Automatic Test Markup Language Draft, December 2004.

[19] RosettaNet, RosettaNet Implementation Framework 2.0, February 2000.

[21] W3C, Hypertext Transfer Protocol – HTTP/1.1 (RFC 2616), June 1999.

[22] IETF, Simple Mail Transfer Protocol (RFC 2821), April 2001.

[23] OASIS, ebXML Implementation, Interoperability and Conformance TC, Deployment Profile Templates V1.1, October 2006.

[24] UN/CEFACT, Core Components Technical Specification – Part 8 of the ebXML Framework, V2.01, November 2003.

Copyright © OASIS Open 2006. All Rights Reserved.

9.Appendix: Notices

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's procedures with respect to rights in OASIS specifications can be found at 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 specification, can be obtained from the OASIS Executive Director.

OASIS invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to implement this specification. Please address the information to the OASIS Executive Director.

Copyright © OASIS Open 2005-2006. *All Rights Reserved.*

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 paragraph are included on all such copies and derivative works. However, this document itself does not be modified in any way, such as by removing the copyright notice or references to OASIS, except as needed for the purpose of developing OASIS specifications, in which case the procedures for copyrights defined in the OASIS Intellectual Property Rights document 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 RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.