XML Testing and Event-driven Monitoring of Processes (XTemp) Version 1.0

Committee Specification 01

07 December 2011

Specification URIs

This version:

http://docs.oasis-open.org/tamie/xtemp/v1.0/cs01/xtemp-v1.0-cs01.odt (Authoritative)
http://docs.oasis-open.org/tamie/xtemp/v1.0/cs01/xtemp-v1.0-cs01.html
http://docs.oasis-open.org/tamie/xtemp/v1.0/cs01/xtemp-v1.0-cs01.pdf

Previous version:

N/A

Latest version:

http://docs.oasis-open.org/tamie/xtemp/v1.0/xtemp-v1.0.odt (Authoritative)
http://docs.oasis-open.org/tamie/xtemp/v1.0/xtemp-v1.0.html
http://docs.oasis-open.org/tamie/xtemp/v1.0/xtemp-v1.0.pdf

Technical Committee:

OASIS Testing and Monitoring Internet Exchanges (TaMIE) TC

Chairs:

Hyunbo Cho (hyunbo.cho@gmail.com), Individual
Jacques Durand (jdurand@us.fujitsu.com), Fujitsu Limited

Editors:

Jacques Durand (jdurand@us.fujitsu.com), Fujitsu Limited
Stephen D. Green (stephengreenubl@gmail.com), Individual

Additional artifacts:

This prose specification is one component of a Work Product which also includes:

- XML schema: http://docs.oasis-open.org/tamie/xtemp/v1.0/cs01/xsd/xtemp.xsd

Declared XML namespaces:

- http://docs.oasis-open.org/tamie/xtemp/200909

Abstract:

XTemp is an XML mark-up language that is event-centric and intended for the analysis of a sequence of events that represent traces of business processes. It is designed for both log analysis and real-time execution. It leverages XPath and XSLT.

Status:

This document was last revised or approved by the OASIS Testing and Monitoring Internet Exchanges (TaMIE) TC on the above date. The level of approval is also listed above. Check the "Latest version" location noted above for possible later revisions of this document.

Technical Committee members should send comments on this Work Product to the Technical Committee's email list. Others should send comments to the Technical Committee by using the

"Send A Comment" button on the Technical Committee's web page at http://www.oasis-open.org/committees/tamie/.

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

Citation format:

When referencing this Work Product the following citation format should be used:

[XTemp-V1.0]

XML Testing and Event-driven Monitoring of Processes (XTemp) Version 1.0. 07 December 2011. OASIS Committee Specification 01.
http://docs.oasis-open.org/tamie/xtemp/v1.0/cs01/xtemp-v1.0-cs01.html

Notices

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

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

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

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

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

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

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

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

Table of Contents

1 Introduction

This document is a specification of the XTemp language for event-driven analysis, testing and monitoring of business and application processes, as well as of transactions between business partners. It is intended to be used in environments where the output of such processes can be traced and stored as events, for which XML is the best supported – but not exclusive - representation. XTemp is an XML markup that leverages existing XML processing tools for its execution.

Two requirements motivate a standard representation of test scripts:

- 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 engine, 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 - hence standardized - from one test engine to the other. This requires a processable representation, at higher level than programming languages, as well as independent from other mark-ups used for controlling some of the layers of e-Business (Web services choreography, business transactions...) as implementations of these may be themselves the target for testing.

Organization of the document:

Section 2 describes the general objectives of the language.

Section 3 describes the general design and semantics of the language

Section 4 describes the syntax and detailed semantics of the language.

Section 5 describes the assumed event model and event management.

Section 6 is the conformance section and defines a core subset of XTemp.

Appendix A: gives two complete worked examples of scripts.

Appendix B and C: give XML schemas for the events board and wrapper, and for xtemp markup language.

1.1 Intended Audience

The primary audience for this document are the users of business processes and the writers of test suites and monitoring scripts for such processes.

1.2 Non-Normative References

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

[CONF2] Conformance testing and Certification Framework (OASIS, Conformance TC, June 2001)
see http://www.oasis-open.org/committees/download.php/309/testing_and_certification_framework.pdf

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

[VAR] Variability in Specifications, WG note (W3C, 2005)
see http://www.w3.org/TR/2005/NOTE-spec-variability-20050831/

[TMD] Test Metadata, QA Interest Group note, (W3C, September 2005)
see http://www.w3.org/TR/2005/NOTE-test-metadata-20050914/

[TAG] Test Assertions Guidelines, (OASIS 2010)
see http://www.w3.org/TR/2005/NOTE-test-metadata-20050914/

2 Objectives and General Approach

2.1 Testing and Monitoring of Business Processes and Transactions

Testing and monitoring of business processes and transactions as well as more generally of systems the behavior of which can be traced by events, fall in three categories:

- Compliance with specifications. Such specifications may be of a business transaction, business process definition, documents exchanged, or of infrastructure behavior (e.g. messaging protocol). Enabling the automatic generation of XTemp scripts from such specifications, when these specifications are formal – e.g. process definition, choreographies, document schema or rules – is part of the requirements although the methodology to achieve this is out of scope of this document. Some test assertion design and best practices, such as those in Test Assertions Guidelines [TAG] may be used for deriving scripts from such representations even when automatic generation is not possible.

- Compliance with agreements. Such agreements may be business agreements such as SLAs, or regulatory compliance rules. They may be infrastructure configuration agreements (e.g. ebXML CPA, WS-Policy). This category of application includes SLA monitoring, business metrics and aspects of business activity monitoring (BAM) that are closest to operations e.g. for regulatory compliance.

- Business Operation intelligence. Such monitoring is not directly related to compliance, but primarily intended for generating reports and various analytics of business activities. This includes analysis of logs of processes and business transactions for reports and BI. This category of application includes BAM (business activity monitoring). In its dynamic aspect, this monitoring is addressing the need for visibility in business processes and service-oriented systems, which includes problem detection/anticipation, diagnostics and alarm generation.

2.2 Live and Deferred Testing

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

From the viewpoint of script execution semantics, "live" and "deferred" are not distinguished: the same script is executable on input that is either live or logged. To ensure flexibility for handling various monitoring contexts and situations, mixing of both execution modes must be supported:

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

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

When events are consumed in a publish-subscribe mode, a simple queuing mechanism is sufficient to provide the above flexibility. However, XTemp must be able to correlate past events. For this reason the notion of event board is necessary, which supports such correlation and querying capability in addition to the queue consumption model.

2.3 An XML-Centric Language for Tests and Events

XTemp is intended as a language for testing and monitoring of applications and business processes, the behavior of which can be traced and monitored via events. The notion of event in XTemp is broad: messages, alarms, monitoring output, configuration files, contracts, may be events. The best supported format for events is XML, as this document leverages XPath as expression language for event selection and correlation. However other event formats may be supported if the appropriate expression language is implemented. For example, EDI can be supported either by providing an XML mapping of such events, or by using EDIpath instead of XPath. There are three levels at which XML is leveraged:

- XML as test scripting format (I.e.. XML mark-up for test scripts definitions, XPath expressions and Xpath libraries for test case logic) ,
- XML as base format for events. WThese events may in turn represent all kinds of objects - alarms, message items, logs, configuration documents, metadata definitions. Some of these may not be natively in XML but can be wrapped with XML.
- XML as report format for test reports and monitoring reports (e.g. structured report data and metadata for subsequent HTML rendering).

3 General Language Semantics

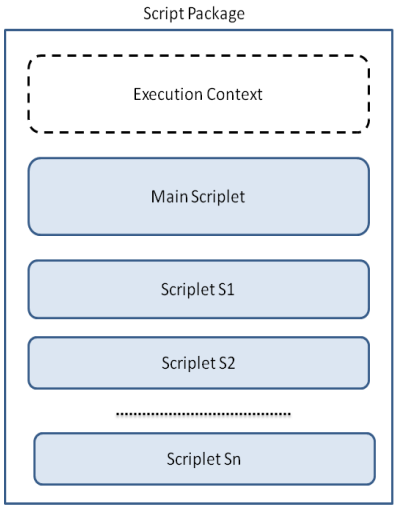
3.1 Some Execution and Usage Aspects

3.1.1 High-level Design

The XTemp language is designed for testing and monitoring processes or business transactions of various kinds, and more particularly for analyzing and validating event patterns that are generated by these processes. To this extent, XTemp may be described as an event-processing language. The content and structure of these events may be quite diverse, but a common XML wrapper is assumed (see section 5).

The top-level constructs are the script package and the scriplet:

- The script package, or “script”: This is the main unit of execution. The script package contains an “execution context” (<execution-context> element) that defines various global constructs and bindings - e.g. for declaring event boards. The script package contains also zero or more "scriplets". The execution context in a script package defines which scriplet to start execution with - or main scriplet. In case other scriplets are defined, the main scriplet is expected to invoke these directly or indirectly.
- The scriplet: A scriplet defines a flow (or thread) of execution for a sequence of atomic operations. Scriplets can execute concurrently or not (see detailed meaning in the next section), and can be started in a blocking or non-blocking way.



A script package or script terminates its execution when the execution flow of its scriplets terminates (either on normal termination, or by exiting).

XTemp is designed so that it leverages existing XML processing languages for special features such as logical conditions and event selection. The default language for all logical expressions over XML content is XPath, along with its existing function libraries (e.g. advanced set of functions for time and duration management). An XTemp implementation is therefore designed as a "layered" language :

- Layer 1: an XML markup (XTemp) for top-level constructs and controls.
- Layer 2: an expression language for embedded expressions (conditions, XML fragments...), by default XPath2.0.

XTemp (layer 1) is open to other XML dialects than XPath1.0 or XPath2.0 in layer 2, e.g. it can use XQuery instead of or in combination with XPath. Another XML dialect is XSLT, which is the default dialect for writing external functions.

A base subset of XTemp (“core” XTemp) is defined, that can be implemented entirely in XSLT2.0.

3.1.2 Inputs and Outputs

The only inputs of an XTemp script are events. Event sources are declared as "event boards" in the execution context of script packages. External events are posted to an event board after proper formatting (XTemp-compliant wrapping) by an event adapter.

There are three kinds of outputs for a script package:

(a) Events: A script can generate and post events to an event board. Generated events use by default the XTemp event format. A log of such events may be the preferred format for results of monitoring or testing. Outputs in form of events also allow for dynamic coordination between concurrent scriplets, as well as pipe-line processing of test material by a chain of scripts.

(b) X-effects: A script can produce an XML document called an "X-effect" (or XML side-effect). The X-effect of a script is the result of composing the individual X-effects of each statement (if any), rather than of a particular output function. An X-effect may also be partially stated inline of scriplets, as XML fragments with foreign namespaces. Some script packages may be designed so they resemble more a parameterized XML document, others may produce very terse or no X-effect. Typically, the X-effect is a convenient output to produce a test report or monitoring report, e.g. is easy to use as a starting point for HTML rendering.

(c) Standard output: A script can write “messages” to the standard output stream. This is helpful for debugging and for getting dynamic status information.

3.1.3 Virtual Present Time and Modes of Execution

Flexibility in alternating "live" and "deferred" execution modes during the same execution is obtained using a notion of virtual present time (VP-time) which can be same as or different from the actual execution time. When a script starts executing, it is given a starting date/time which is its VP-time.

- Live execution: the VP-time is set to the actual present time (or “AP-time”), which is the default. In this case events will be caught as they occur: every “catch” operation is "waiting" for these occurrences. These events are still transiting by the event board, acting as a publish-subscribe device.
- Deferred execution: the VP-time is set to a past date/time. In this case it will catch events that are already logged in the event board.

The processing of the script is similar in both cases as the only difference is in the value of the VP-time. All events - live or logged - are still obtained from the event board. Only in some cases, the execution of the "catch" operation will need to wait for events to be logged to the board, and in other cases will not as they already are on the board. As previously mentioned, a script execution may mix live and deferred modes.

3.2 Virtual Present Time and Concurrency Semantics

3.2.1 Concurrency in XTemp

The concept of concurrency in XTemp is entirely dependent on the notion of “virtual present time” (VP-time. When a scriplet starts to execute, it is assigned a VP-time that will condition its event consumption and will timestamp its event production. The default VP-time assignments are:

- The first scriplet of a script package is assigned the initial VP-time of this script, the default of which is in turn the actual present time (AP-time).
- The VP-time of a scriplet S2 started by a scriplet S1, is the value of VP-time in S1 when [start S2] is executed.

These default values can be override by the <start> operation, which allows to set the VP-time of the started scriplet (see the start/@vptset attribute in section 4). Inside a scriplet, the VP-time may be progressed by two operations:

- <wait> : will add some predefined duration to the VP-time, or wait until some date, or yet until some other scriplets complete.
- <catch> : when waiting - virtually or really - for some event to occur, will advance the VP-time to the occurring date of events being caught. Event catching in a scriplet is only considering (by default) events occurring at or after the current VP-time.

Besides <wait> and <catch>, the execution duration of other XTemp operations is considered as negligible as far as the VP-time is concerned: in other words, these operations do not affect the VP-time. The *VP-time window* of a scriplet execution is defined as the [starting VP-time, ending VP-time] time interval of the execution. Intuitively, concurrent execution is achieved when the VP-time windows of two scriplets overlap.

IMPORTANT NOTE: In this document, *concurrency* must not be understood as requiring multi-threading or actual parallel execution. The meaning of this term is here entirely defined in terms of VP-time windows of respective scriplets. In practice, two “concurrent” scriplets that do not communicate with or affect each other – i.e. scriplets that do not “mask” events nor generate events - could still execute one after the other - i.e. in an entirely serialized way - with same result as if actually executed concurrently. Concurrency is entirely determined by the start VP-time assigned to these scriplets – e.g. two scriplets given the same starting VP-time will be considered here as concurrent, regardless of their actual execution times.

By default, events consumed by a scriplet are still “visible” and accessible by another scriplet (unless they are masked, see <mask>).

3.2.2 Blocking vs. Non-Blocking Scriplet Execution

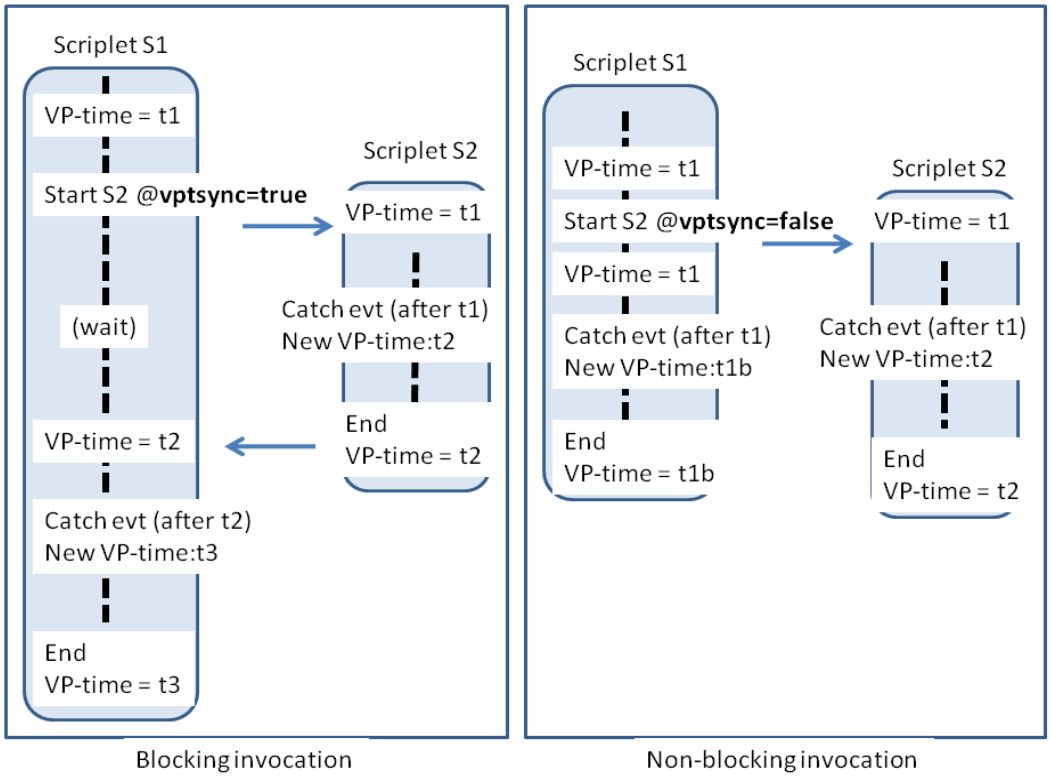
There are two modes of scriplet invocation. When a scriplet S1 starts a scriplet S2, it can do so either in a blocking or non-blocking way:

- Blocking invocation: intuitively, the invoking scriplet (S1) will wait until the invoked scriplet (S2) terminates. The next statement in the invoking scriplet S1 (after <start>), will execute at a VP-time that is same as the VP-time set at the end of the invoked scriplet (S2). In other words, the VP-times of S1 and S2 are “synchronized” after S2 completes (see start/@vptsync="true" in Section 4). More generally, to accommodate the case where a starting time is set at a date/time anterior to the invoking time (@vptset="a past date/time”) the VP-time of the next statement in S1 is either the last VP-time value of S2 or the last VP-time value in S1 (just before invoking S2), whichever occurs latest.
- Non-blocking invocation: intuitively, the invoked scriplet (S2) will not cause the invoking scriplet (S1) to wait. In other words, the VP-times of S1 and S2 are not “synchronized” after S2 completes (see start/@vptsync="false" in Section 4). The next statement in the invoking scriplet S1, will execute at a VP-time that is same as the VP-time value just before executing the <start> statement, this regardless of the value of start/@vptset.

Non-blocking invocations should not be seen as only useful for actual concurrent (or multi-threaded) processing. In many cases, it makes scripting easier and more intuitive, even when execution is entirely deferred on past (logged) events that could otherwise be processed serially in a single-threaded way.

Various cases of blocking and non-blocking invocations are illustrated below.

The following figure illustrates both modes of scriplet invocation, and how the VP-time is affected – or not - in the invoking scriplet.

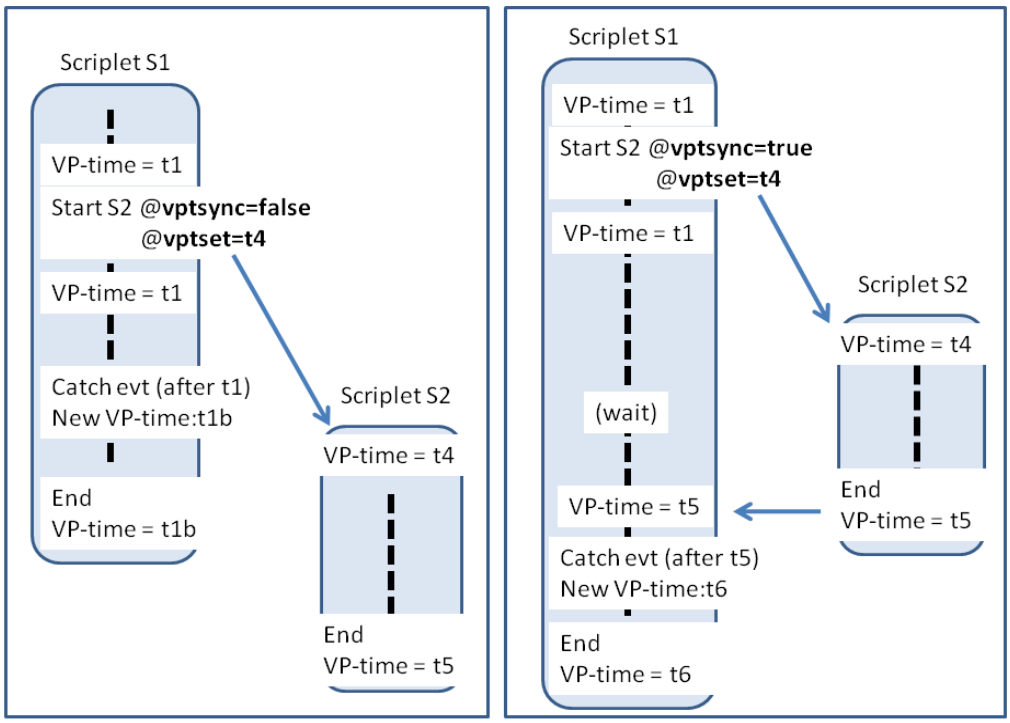


Blocking invocation case: When a scriplet S1 does a blocking invocation of a scriplet S2, the VP-time of the invoked scriplet S2 is initiated at the current VP-time of the invoking scriplet S1 (unless a different VP-time value is given using start/@vptset, as illustrated in the next figures). The scriplet S1 is then "blocked" until the VP-time at the end of S2 is known and assigned as current VP-time in S1.

Non-blocking invocation case: In the non-blocking invocation (see above), S1 will ignore the ending time of S2. A single-threaded execution may still execute S2 first before executing the next statement in S1. The execution semantics would still allow "concurrent" catching (in terms of virtual present) of same events by S1 and S2, as both will select events starting from VP-time t1. In the above figure, S1 is catching an event at time t1b while S2 is catching an event at time t2. Depending on their respective selection expressions, these catches could capture the same event or different ones, causing the new VP-time in S1 (t1b) to be either prior or after the new VP-time in S2 (t2).

Note: In a single-threaded execution of the non-blocking case that started "live" (VP-time = present time), S2 could be executed first live, then the remaining part of S1 can be executed "deferred" on the log of events, starting from time t1 now in the past. Clearly, more fine-grain serialization of S1 and S2 executions would be required if these two scriplets communicate with each other, e.g. if S1 is consuming an event posted by S2 or vice-versa.

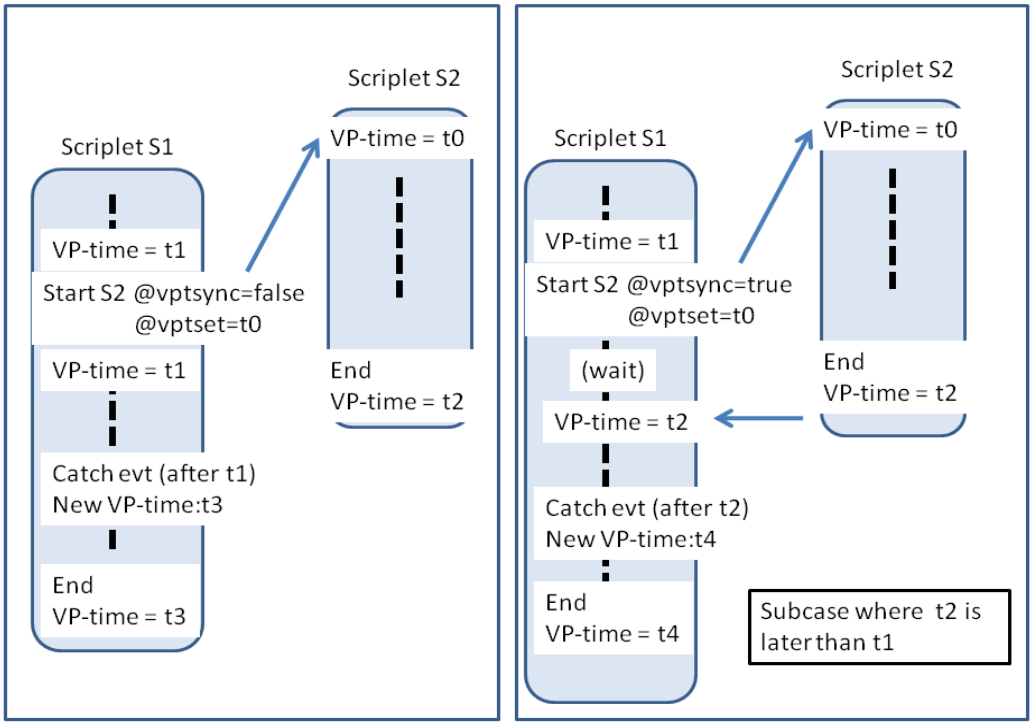
The following figure illustrates the two modes of invocation, when setting the VP-time of the invoked scriplet S2 to a future time ("forward" invocations). In the case of non-blocking invocation, the invoking scriplet (S1) may terminate before the invoked scriplet (S2). This is never the case with blocking invocation.



Forward non-blocking invocation

Forward blocking invocation

The following figure illustrates the two modes of invocation, when setting the VP-time of the invoked scriplet to a past time relative to the invocation time ("backward" invocations).

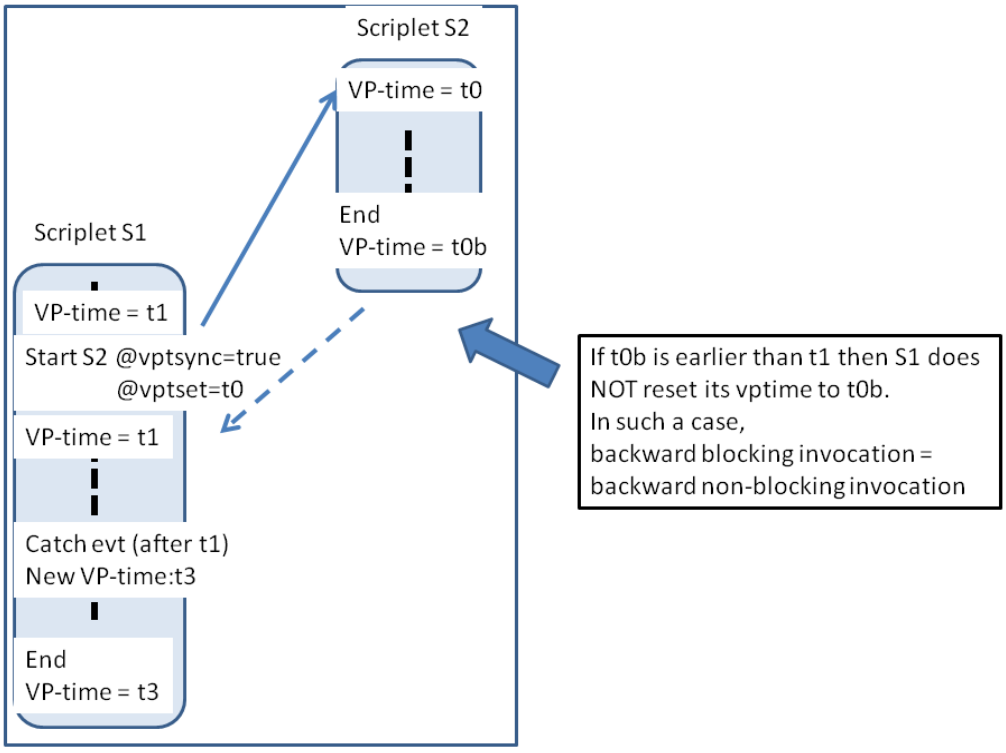


Backward asynchronized invocation

Backward synchronized invocation

In case the scriplet (S2) is invoked in a backward blocking way, two sub-cases need be considered: the above figure shows the case where the termination of S2 is at a later date/time than the invocation VP-time (t1) in the invoking scriplet (S1). This case involves actual "blocking" of the invoking scriplet, as when

@vptset is not used. The case where S2 termination time (t0b below) is still earlier than the invocation time (t1) does not lead to a resetting of VP-time in S1, as illustrated below:



"Fully past" backward blocking invocation

Such an invocation is said to be "fully past" relative to the VP-time of the invoking scriplet.

The following general rules summarize the VP-time semantics for scriplet executions:

Rule 1: *The VP-time for the next statement of an invoking scriplet immediately following a non-blocking invocation is always the VP-time of invocation in the invoking scriplet.*

Rule 2: *The VP-time for the next statement of an invoking scriplet immediately following a blocking invocation is always the latest date/time of (1)VP-time of invocation in the invoking scriplet, (2)termination VP-time in the invoked scriplet.*

Rule 3: *The VP-time in a scriplet can only monotonically increase.*

Even when the VP-time can be set to an earlier date/time for some operations (start, catch), once these operations are complete the VP-time in the performing scriplet is always equal or greater after the operation than before.

In terms of concurrency semantics, XTemp cannot be considered an Actor language: Scriplets are not aware of each other and do not communicate explicitly with each other. The model is rather of CSP (Communicating Sequential Processes), where scriplets behave as processes, consuming events and communicating via events only - except when a scriplet is starting another scriplet, in which case parameters may be passed.

4 Language Constructs

4.1 Scriplets

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

4.1.1 xtemp:scriplet

Role: A scriplet is the basic unit of script. In addition to representing a flow of execution for XTemp statements, a scriplet has the following semantics:

(a) Thread semantics: a scriplet executes its operations in one thread. However, a scriplet can invoke another scriplet "concurrently" or more accurately in a non-blocking way.

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

(c) Scoping unit for variables declarations.

Compact RelaxNG notation:

(default namespace = "http://docs.oasis-open.org/tamie/xtemp/200909")

```
element xtemp:scriplet { scriplet_body }

scriplet_body =
       attribute name { xsd:NCName },
       element xtemp:param { param_body }*,
       ( xtempstatement | xeffectitem )*

xeffectitem =              # any non-xtemp qualified element
  element *:* - xtemp:* { attribute * { xsd:normalizedString } *,
       (xtempstatement | xeffectitem)* }

NOTES:
-   the above means any element with a namespace other than the XTemp namespace,
    and containing in turn a sequence possibly empty of mixed XTemp statements and
    other X-effects.


xtempstatement =
  (
       element xtemp:var { var_body } |
       element xtemp:mask { mask_body } |
       element xtemp:catch { catch_body } |
       element xtemp:decide { decide_body } |
       element xtemp:eval { eval_body } |
       element xtemp:exit { exit_body } |
       element xtemp:if { if_body } |
       element xtemp:loop { loop_body } |
       element xtemp:message { message_body } |
       element xtemp:post { post_body } |
       element xtemp:wait { wait_body } |
       element xtemp:start { start_body } |
       element xtemp:call-adapter { calladapter_body }
  )
```

X-effect of a scriplet:

The X-effect of a scriplet is the concatenation of the X-effects of its child elements:

```
X-effect( scriplet ) =
for-each top-level statement in scriplet body
  { X-effect( statement )  }
```

NOTE: the above is an algorithmic notation only intended to describe the semantics of calculating X-effects. It has no relation to the XTemp syntax.

4.1.2 xtemp:param

Role: defines a parameter for a scriplet.

Compact RelaxNG notation:

```
element xtemp:param { param_body }

param_body =
       attribute name {xsd:NCName} ,
       attribute type {xsd:NCName} ?
       text
```

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

X-effect of param:

There is no X-effect produced by executing the param statement.

4.1.3 xtemp:start

Role: start a scriplet execution.

Compact RelaxNG notation:

```
element xtemp:start { start_body }

start_body =
       attribute scriplet {xsd:NCName} ,
       attribute vptsync {xsd:boolean} ? ,
       attribute vptset {xsd:normalizedString } ? ,
       attribute bubble-exit {xsd:boolean} ? ,
       attribute group {xsd:NCName} ?,
       element  xtemp:with-param { withparam_body } *

withparam_body =
       attribute name {xsd:NCName } ,
       attribute expr {xsd:normalizedString} ? ,
       mixed { xeffectitem }
```

Semantics:

Invokes a scriplet with the possibility to pass some parameters and to decide how the invoked scriplet will affect the execution of the invoking scriplet when returning, e.g. regarding exit semantics and VP-time management.

scriplet: the name of the invoked scriplet, which must belong to the same scriplet package as the invoking scriplet.

vptset: (optional) the virtual present time (VP-time) to be used when starting the scriplet. By default, this starting date is the VP-time of the invoking scriplet. Allows for setting a date different from the default, e.g. a past date that allows catching of past events. The value must be either a date/time constant (e.g. <start vptset="2011-10-12T15:12:00" ...>) or an expression that resolves into a date/time, in which case the expression must be enclosed in {} (e.g. vptset="{$mystartime}" or vptset="{xsd:dateTime($startime + $someduration)}").

vptsync: (optional) Indicates the mode of execution of the invoked scriplet - blocking or not - relative to the invoking scriplet. A scriplet S1 may invoke a scriplet S2 either in a synchronized or non-synchronized way:

- when @vptsync="true" (default value), the invocation of S2 is blocking the invoking scriplet. This means that the next statement in the invoking scriplet S1 (after <start>), will execute at a VP-time that is same as the VP-time set at the end of the invoked scriplet. More generally, to accommodate the case where a starting time is set at a date/time anterior to the invoking time (vptset="past date/time") the VP-time of the next statement in S1 is either the last VP-time value of S2 or the last VP-time value in S1 (just before invoking S2), whichever occurs latest..
- when @vptsync="false", the invocation of S2 is not blocking the invoking scriplet S1.This means that the next statement in the invoking scriplet S1, will execute at a VP-time that is same as the VP-time value just before executing the <start> statement, this regardless of how @vptset is used.

bubble-exit: (optional) if "true" any exiting from the invoked scriplet, will bubble-up in the invoking scriplet, causing its termination when the <start> statement completes. If "false" (default) the invoking scriplet will resume its execution at the next statement after the <start> statement, regardless of the way the invoked scriplet terminated.

group: defines the group of executions to which this dynamic scriplet execution must belong. This group may be used later to join these scriplet executions (using <wait>).

X-effect of a start:

The X-effect of a start statement is the X-effect of the execution of the started scriplet.

4.2 Conditional Statements

Conditional statements are of two kinds:

(a) simple conditions (xtemp:if)

(b) switch conditions (xtemp:decide)

4.2.1 xtemp:if

Role: define a simple conditional statement.

Compact RelaxNG notation:

```
element xtemp:if { if_body }

if_body =
       attribute expr {xsd:normalizedString} ,
       ( xtempstatement | xeffectitem ) *
```

Abstract example:

```
<if expr="(some XPath boolean expression)">
   (body of xtemp statements and/or of foreign xml fragments)
</if>
```

Semantics:

The body of the <if> element is executed only if the condition stated in the @expr attribute is true. The body may be a sequence of one or more of the following: an XTemp statement or an inline X-effect.

X-effect of a simple condition:

The X-effect of a simple condition is evaluated as follows:

```
if (expr = true) then
  X-effect( body )
else nil
endif
```

4.2.2 xtemp:decide

Role: define a switch condition, with multiple tests and outcomes and one default outcome.

Compact RelaxNG notation:

```
element xtemp:decide { decide_body }

decide_body =
       element xtemp:if {if_body} ,
       element xtemp:else-if {elseif_body} *,
       element xtemp:else {else_body} ?

elseif_body =
       attribute expr {xsd:normalizedString} ,
       ( xtempstatement | xeffectitem ) *

else_body =
               ( xtempstatement | xeffectitem ) *
```

Abstract example:

```
<decide>
   <if expr="(bool_expr1)">
      (body-1)
   </if>
   <else-if expr="(bool_expr2)">
      (body-2 )
   </else-if>
   <else>
      (body-last )
   </else>
</decide>
```

Semantics:

The embedded "if" statement is evaluated as previously defined. In case its conditional expression failed, the first "else-if" statement (if any) is executed, with a similar semantics as an "if" statement. In case its conditional expression failed, the next "else-if" statement is executed (if any), and so on. In case all previous conditional expressions failed, the "else" statement (if any) is executed, meaning its body is executed.

X-effect of a switch condition:

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

```
if ( bool_expr1 = true) then
  X-effect( body-1 )
else if ( bool_expr2 = true) then
       X-effect( body-2 )
     else
       X-effect( body-last )
     endif
endif
```

4.3 Variables

4.3.1 xtemp:var

Role: define a variable and assign a value.

Compact RelaxNG notation:

```
element xtemp:var { var_body }

var_body =
       attribute name {xsd:NCName} ,
       attribute type {xsd:NCName} ? ,
       attribute expr {xsd:normalizedString} ? ,
       mixed {( xtempstatement | xeffectitem ) *,}
```

Semantics:

Variables are typically assigned once within a scriplet. Their visibiity and usability scope is the scriplet when declared at the beginning of a scriplet. They may be declared inside statements (e.g. in the “body” of an xtemp:if) , in which case their visibility is the script embedded in the statement. If scriplet "S1" invokes scriplet "S2" and needs to make a variable V1 usable by "S2", it uses parameters (V1 value is passed as parameter of S2). A variable can be made global to the script package by declaring it in the execution context.

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

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

The optional @type attribute value indicates the value type which can be:

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

For types other than 'xml', the @type attribute is necessary if the variable is to be used in an expression that expects typed values - unless some explicit casting function is used in the expression.

Variables may be assigned XML mixed content, i.e. mixing text nodes and XML elements.

Variable assignment (or rather, initialization) is the following ways:

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

Examples:

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

```
<var name="greeting" type="string">hello</var>
```

```
<var name="minprice" type="double">99.50</var>
```

```
<var name="timeout" type="date">...</var>
```

```
<var name="timeout" type="xml">
       <myapp:po>
      <myapp:ref>123456</myapp:ref>
      <myapp:product>greenbike</myapp:product>
      <myapp:qty>2</myapp:qty>
      <myapp:price>165</myapp:price>
    </myapp:po>
</var>
```

(2) by enclosing one or more XTemp statements.

The semantics of such an assignment is that the X-effect of these XTemp statements is assigned to the variable.

Example:

```
<var name="PO" type="date">
   <xtemp:eval expr="$PO/creationdate"/>
</var>
```

(3) by a combination of both, i.e. the body of the variable definition is a mix of X-effect and xtemp statements.

Variable declarations are the only way to capture the output (i.e. its X-effect) of a scriplet invocation, for further processing. This is done by embedding the scriplet invocation <xtemp:start> in the body of the variable definition:

Example:

```
<var name="PO" type="xml">
       <start scriplet="addpo">
              <with-param name="mypo" expr="$mynewpo"/>
              <with-param name="previoustotal" expr="$totalpos"/>
              <with-param name="mydiscountrate" expr="$mydiscountrate"/>
       </start>
</var>
```

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

Example:

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

The above variable $PO will only be assigned when the xtemp:catch statement completes. If the catch is successful, $PO will contain one or more events.

As a general rule, the XML enclosure (XML child element(s)) of an <var> element is always processed in the same way as it would if enclosed in a scriplet definition, and its X-effect (see Section ...) is assigned to the variable.

In the following example, the totalamount element in the purchase order is "parameterized" by including an <eval> element that will produce another variable value ($total) when the variable $PO is being assigned at execution time:

```
<var name="PO">
   <myapp:PurchaseOrder ref="1234">
      <myapp:Item>abc</myapp:Item>
      <myapp:totalamount>
         <xtemp:eval expr="$total"/>
      </myapp:totalamount>
   </myapp:PurchaseOrder>
</var>
```

Assigning the result of an expression - e.g. written using XPath - is done using the statement **xtemp:eval**.

The following variable assignment when executed after the previous example, will give $POamount the same value as the variable $total:

```
<var name="POamount" type="int">
  <eval type="XPath2" expr="$PO/myapp:PurchaseOrder/myapp:totalamount"/>
</var>
```

It is equivalent to the shorter notation:

```
<var name="POamount" type="int"
expr="$PO/myapp:PurchaseOrder/myapp:totalamount"/>
```

The elements inside the purchase order wouldn't have to be limited to literal XML elements and <eval>. Other Xtemp statements can be used such as <if>:

```
<var name="PO">
   <myapp:PurchaseOrder ref="1234">
      <myapp:Item>abc</myapp:Item>
      <myapp:totalamount>
         <xtemp:eval expr="$total"/>
      </myapp:totalamount>
      <if expr="$physicalItem">
```



```
        <myapp:weight>
         <xtemp:eval expr="$weight"/>
        </myapp:weight>
      </if>
   </myapp:PurchaseOrder>
</var>
```

Reserved variables:

$currentvpt: this variable contains the current VP-time (date/time) of a scriplet execution.

X-effect of a variable declaration:

The X-effect of a variable declaration is null. Indeed, the X-effect of the XML enclosure of a variable definition is captured as the value of this variable, and this assignment in itself is not producing any X-effect. Effecting the value of this variable may be done by using it afterward inside an eval statement: e.g. <xtemp:eval expr="$PO"/>

4.4 The X-effect of Various Constructs

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

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

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

Each statement or body of statements has an X-effect (possibly empty). However, not all X-effects will be produced as actual output of a script execution. For example some X-effects are captured for internal use, by variable declarations. The actual X-effect resulting from the execution of a script package, is called the "output" X-effect. The output X-effect is the final trace of this execution as it appears in the output document.

4.4.1 In-line X-effects

The X-effect to be produced by a scriplet may be specified in-line, i.e. by inline inserts of externally-namespaced XML fragments inside the scriplet. An XML fragment is externally-namespaced if it has a namespace different from the xtemp namespace, and if this namespace is declared in the script package.

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

4.4.2 Inserting the value of expressions in X-effects: eval

Compact RelaxNG notation:

```
element xtemp:eval {
       attribute lg {xsd:NCName} ? ,
       attribute expr {xsd:normalizedString}
}
```

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

Setting the value of an XML attribute is done as in XSLT by enclosing the nexpression using {}:

```
      <myreport myattr="{$myvar/*:outer/*:inner[2]}">
        <xtemp:eval expr="$myvar/*:outer/*:inner[2]"/>
      </myreport>
```

In the above example, the element myreport will have same value as the attribute @myattr.

4.4.3 X-effect Concurrency Semantics

The combined X-effect of scriplets where one is started by the other, is following these rules:

Rule 1. When scriplet S1 "starts" (either blocking or non-blocking) scriplet S2, the X-effect of S1 is obtained by :

(1) capturing the X-effect of S2 at the end of S2 execution,

(2) substituting the <starts scriplet="S2" vptsync="..." /> statement in S1 with the X-effect of S2.

(3) generating the final X-effect of S1 at the end of its execution, after substitution in (2).

This rule applies recursively to all scriplet invocation trees, possibly combining blocking and non-blocking invocations.

4.5 Exiting

The <exit> statement allows for interrupting the execution of a scriplet before its normal ending (execution of its final statement). The statement immediately causes the termination of the containing scriplet, except when the exit is occurring inside a variable definition (see later).

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

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

When a scriplet exits, the X-effect of the scriplet, as produced until execution of the <exit>, is preserved. In other words, there is no undoing of the current effect. When exiting, the last X-effect increment consists of inserting the <exit> X-effect.

Exiting is not always used as an "exception" mechanism (abnormal termination). It is often a scripting convenience for terminating a test case in various ways without having to use a cascade of conditional statements. In such cases, the X-effect built so far before and on top of the exiting statement, needs be recorded.

4.5.1 xtemp:exit

Role: exits from the current scriplet and terminate the current scriplet execution, except when embedded in a variable declaration.

Compact RelaxNG notation:

```
element xtemp:exit { exit_body }

exit_body =
       attribute message {xsd:normalizedString} ? ,
       attribute result {xsd:normalizedString} ? ,
       mixed {( xeffectitem ) *,}
```

In the following example, if S3 exits then S2 will still execute the condition that will produce the xml fragment `<myapp:outcome>S2 terminated normally </myapp:outcome>`. The overall X-effect of S2 would be:

```
<myapp:report>
       <myapp:outcome>S2 terminated normally</myapp:outcome>
</myapp:report>
```

In the presence of start/@bubble-exit="true" (<start scriplet="S3" bubble-exit="true"/>) an exit in S3 would propagate to S2 and the conditional statement in S2 would not execute - the overall X-effect of S2 would then be the element:

<myapp:report>

<myapp:exiting>from S3 </myapp:exiting>

</myapp:report>:

```
<scriplet name="S2">
       <myapp:report>
  <start scriplet="S3"/>

  <if expr="fn:true()">
    <myapp:outcome>S2 terminated normally</myapp:outcome>
  </if>
  </myapp:report>
</scriplet>

<scriplet name="S3">
...
  <if expr="...">
    <exit>
              <myapp:exiting>from S3</myapp:exiting>
       </exit>
  </if>
...
</scriplet>
```

When a scriplet S1 starts a scriplet S2 in a non-blocking way, with @bubble-exit="true", then the invoking scriplet will be interrupted whenever the invoked scriplet S2 is exiting, i.e. one of the following occurs:

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

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

X-effect of Exit:

The X-effect of <exit> is the X-effect of the contained XML fragment(s), which may in turn contain embedded XTemp statements.

4.5.2 Exiting from Variable definitions

When the body of a variable definition "exits", the variable definition terminates immediately but the exiting does not bubble-up to the embedding scriplet.

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

```
<scriplet name="S1">
  <var name="v3">
    <if expr="$somevar = 0">
      <exit/>
    </if>
    <catch> <match event="E1">...</match></catch>
  </var>
  <if expr="$v3">
    <exit/>
  </if>
</scriplet>
```

In order to escalate (bubble-up) the exiting beyond the variable - e.g. in above example, in order to exit from the scriplet S1 - one would have to add a test on the value of the variable that will perform the exit at scriplet level, as done in above example.

4.5.3 Exiting from Conditional Statements

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

(a) as a leaf of a conditional tree (e.g. <else-if expr="..."><exit/></else-if>), in which case the exiting is terminating the conditional execution, and also terminating the body of statements within which this conditional statement appears. In other words, an <exit> is always "bubbling-up" at the level above the conditional statement.

(b) as a statement inside a body of statements that is itself a leaf in a conditional tree. In that case, any statement in this body after the <exit> is not executing, and the conditional is terminating as described in (a).

4.6 Iterations

4.6.1 xtemp:loop

Role: define iterative processing of a sequence of XTemp statements.

Compact RelaxNG notation:

```
element xtemp:loop { loop_body }

loop_body =
       element xtemp:on-start {onstart_body} ? ,
```



```
      ( xtempstatement | xeffectitem | until_stmt) *
      element xtemp:lvar-next {var_body} * ,
      element xtemp:on-final {onfinal_body} ?

onstart_body =
      element xtemp:lvar { var_body } *

until_stmt =
      element xtemp:until {
            attribute expr { xsd:normalizedString }
      }

onfinal_body =
      ( xtempstatement | xeffectitem ) *
```

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

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

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

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

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

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

(a) **loop variable**: declared and initialized once using <xtemp:lvar> in the <xtemp:on-start> statement, such a variable keeps its value throughout the loop iterations, but it may be updated from one iteration to the other by re-assigning it at the end of the loop body (<xtemp:lvar-next> statement) e.g. based on its value from the previous iteration. These variables are also in scope of the <on-final> statement.

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

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

The loop element may contain the following child elements (all are optional), in addition to regular XTemp statements inside the loop body:

- An **<on-start>** element: any initialization operation that must be done just once before the loop iterates, is defined in this element. It must be the first child element of the <loop> element. This includes declaration of "loop variables".
- An **<on-final>** element: it contains operations that need be done once when the loop completes, either normally or as result of a break statement. If any aggregated result needs to be produced at the end of the iteration, this is where its computation is final. It must be the last child element of the <loop> element.
- An **<until>** statement. This defines the exit condition for the iteration. On execution, and if the condition is satisfied, the <on-final> element is executed. In such a case, the elements in the body at same level as <until> and after it are not executed (except for <on-final>). There MUST be at least one <untiil> statement in a loop.

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

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

X-effect of a loop:

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

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

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

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

Example 1:

```
 <loop>
        <on-start>
            <lvar name="myitems" expr="$PO/lineItems/item"/>
            <lvar name="item" type="int">1</lvar>
            <lvar name="pototal" type="int">0</lvar>
        </on-start>
        <until expr="mycondition"/>
        <var name="myItemAmount"><eval expr="$myitems[$item]/unitprice *
$myitems[$item]/quantity"/></var>
        <var name="itemRebate">
            <decide>
                <if expr="$myItemAmount gt 1000">
                    <eval expr="$myItemAmount * 0.10"/>
                </if>
                <else><eval expr="0"/></else>
            </decide>
        </var>
        <lvar-next name="pototal" expr="$pototal + ($myItemAmount -
$itemRebate) "/>
        <lvar-next name="item" expr="$item + 1"/>
        <on-final>
            <eval expr="$pototal"/>
        </on-final>
    </loop>
```

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

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

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

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

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

Example 2:

```
 <loop>
            <on-start>
                <lvar name="myitems" expr="$PO/lineItems/item"/>
                <lvar name="item" type="int">1</lvar>
                <lvar name="pototal" type="int">0</lvar>
            </on-start>

            <until expr="$item gt fn:count($myitems)"/>

            <var name="itemSalesEvent">
                <catch>
                    <match>
                        <condition>content/sales/product[@ref =
                            $myitem[$item]/reference]</condition>
                    </match>
                </catch>
            </var>

            <var name="myItemPrice">
                <decide>
                    <if expr="$itemSalesEvent">
                        <eval
expr="$itemSalesEvent/am:content/sales/product/salesprice"/>
                    </if>
                    <else>
                        <eval expr="$myitems[$item]/unitprice"/>
                    </else>
                </decide>
            </var>

            <var name="myItemAmount">
                <eval expr="$myItemPrice * $myitems[$item]/quantity"/>
            </var>

            <var name="itemRebate">
                <decide>
                    <if expr="$myItemAmount gt 1000">
                        <eval expr="$myItemAmount * 0.10"/>
                    </if>
                    <else>
                        <eval expr="0"/>
                    </else>
                </decide>
            </var>

            <lvar-next name="pototal" expr="$pototal + ($myItemAmount -
$itemRebate) "/>
            <lvar-next name="item" expr="$item + 1"/>
            <on-final>
                <eval expr="$pototal"/>
            </on-final>
        </loop>
```


Exiting from inside the body of a loop will exit from the loop without execution of the <on-final> statement, and will also terminate the scriplet executing this loop (except if this loop was inside a variable declaration) in which case only the variable declaration terminates).

4.7 Event Catching

4.7.1 Querying Events vs. Synchronizing with Events

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

The same catch operator is used in two different contexts of event catching:

• **Live execution**,. A live execution operates over "live" event input, i.e. events may not have occurred yet, when the catch starts executing. In that case, "'catch'" is a synchronization operator that will cause a script execution to wait for the expected event(s).

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

However, there is no semantic difference between querying past events and catching "new" events, even in "live" mode: the only difference will be in the effect on the VP-time. A catch operator may both "wait" for some event, then query other (past) events, as the catch operator allows for catching a combination of correlated events. Such a combination is called an "'event pattern"'. Typically, the catch will trigger on some event of the pattern, while other events in the pattern may have already occurred.

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

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

Each "match" statement defines a filter for one of the events of the pattern. Here, the order of these filters is important as it defines the order in which the pattern will be matched. In the above example, a pattern matching will start with the next PO event, then "wait" for the POresponse then the POreceipt events, which are expected to occur later than the PO as stated by the @after attribute.

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

4.7.2 VP-time dependency

The VP-time is conditioning the event catching.

The first event filter (first <match> statement) of a <catch> defines the "**lead event**": it is the first event to be matched for the event pattern defined by this catch statement (e.g. in the previous example, the "PO" event is the lead event). Among the events from the event board that satisfy this first filter, the first lead event to occur that has a timestamp after VP-time, will initiate a pattern matching attempt.

Once an events pattern has successfully matched the <catch> filters, the VP-time is updated to the timestamp of the latest event of the pattern, and the execution control is given back to the next statement after <catch>.

Considering a simple case of a one-event pattern, the following rules decide whether the <catch> is "blocking " or "non-blocking" in the actual execution of a scriplet:

- If VP-time = AP-time (this is a case of "live" execution), then <catch> is blocking the scriplet execution thread until some event(s) is caught that matches its <match> statement(s), or until the @tryfor duration is exhausted.
- If VP-time < (VP-time + tryfor) < AP-time, then <catch> is not blocking the scriplet. This means both VP-time and (VP-time + tryfor) are anterior to AP-time - a case of fully deferred execution. The <catch> statement simply acts as a query over the event board. The VP-time is incremented up to the latest timestamp of this event combination - if such an event(s) is caught before (VP-time + tryfor) - .or is incremented up to VP-time + tryfor if no event satisfies the <catch> in the event board.
- If (VP-time < AP-time < (VP-time + tryfor) , then the <catch> statement may block or not: it starts as a query over the event board up to AP-time, then if no event(s) is selected, blocks the scriplet execution (which becomes "live") until some event(s) is caught that matches its <match> statement(s), or until the @tryfor duration is exhausted.

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

4.7.3 xtemp:catch

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

Compact RelaxNG notation:

```
element xtemp:catch { catch_body }

catch_body =
              attribute tryfor {xsd:normalizedString } ,
              attribute eboard {xsd:NCName} ? ,
              attribute vptset {xsd:normalizedString } ? ,
              attribute vptend {xsd:normalizedString } ? ,
              element  xtemp:match { match_body } *
```

Semantics:

The catch statement is selecting one or more events from the event log, that satisfy a condition specified in the `xtemp:match` subelement(s). In case of a single event catch, the timestamp of the selected event must be same or later than the current VP-time (called the starting VP-time). In case of the selection of a pattern of events, only the lead event must be past the starting VP-time.

- **catch/@eboard**: identifies the event board that is source for the catching. If absent, the "main" or default event board is the default source.
- **catch/@vptset**: sets a starting VP-time for the catch that is different from the current VP-time of the embedding scriplet. This makes it possible to query events that occurred before the scriplet current VP-time, or after a future date/time. After the catch execution, the new VP-time of the executing scriplet will be set based on the latest of :
  - (1)scriplet VP-time before executing catch,
  - (2) date/time of the event (or of the latest event of the pattern])selected by the catch

    In other words, the VP-time of a catch statement is governed by the same rules as the VP-time of a scriplet invoked in a blocking way: the VP-time after the operation can only be later

or same as the VP-time before the operation. The value must be either a date/time constant (e.g. <catch vptset="2011-10-12T15:12:00" ...>) or an expression that resolves into a date/time, in which case the expression must be enclosed in {} (e.g. vptset="{$mystartime}" or vptset="{xsd:dateTime($startime + $someduration)}").

- **catch/@tryfor**: a duration that indicates how long the catch operation may "try" to select an event (or a combination of) that satisfies the defined pattern, thus defining a “catch window”. The value must be either a duration constant (e.g. <catch tryfor="P3DT12H"" ...>) or an expression that resolves into a duration, in which case the expression must be enclosed in {} (e.g. tryfor="{$someduration}" By default, @tryfor is set to the “very long duration” (a configurable, tractable definition of .the “infinite” duration).

- **catch/@vptend**: sets a date/time limit for catching events. It can be seen as an alternative to @tryfor for defining a catch window, and has the same time control semantics. If present with @tryfor, the event catching will be bounded by the earliest date/time between (1) current VP-time + @tryfor, and (2) @vptend. If @vptend is earlier than the starting VP-time (as either defined by @vptset or by the current VP-time if @vptset is absent), it is ignored. By default, @vptend is set to the “very late date” (a configurable, tractable definition of the “infinite” date). The value must be either a date/time constant (e.g. <catch vptend="2009-10-12T15:12:00" ...>) or an expression that resolves into a date/time, in which case the expression must be enclosed in {} (e.g. vptend="{$endperiod}" or vptend="{xsd:dateTime($startperiod + $lengthperiod)}").

- **catch/match**: defines a condition for selecting one event (or a number of events) from the event board. See "match" detailed description.

Execution semantics:

The VP-time at the time the Catch starts executing is called the "initial VP-time".

The datetime: initial VP-time + tryfor, is called the "catch timeout".

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

- the lead event is not previously masked by any scriplet instance of the same package execution instance.

- the lead event timestamp (dateTime) is set after the initial VP-time, and before the catch timeout.

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

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

- the event is not previously masked by any scriplet instance of the same package execution instance.

- the event occurs before the catch timeout (@tryfor)

- the event satisfies the selection condition for this match, which generally correlates with previously selected events. (e.g. via variables defined in preceding "match" siblings).

If there is failure to match an event for this filter - e.g. a POresponse event did not occur before the catch timeout - then the pattern detection resumes to Step 1, for catching the next "lead" event after the timestamp of the previous lead event attempt.

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

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

4.7.4 xtemp:match

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

```
element xtemp:match {  match_body }

match_body =
              attribute event {xsd:NCName } ? ,
              attribute view {xsd:normalizedString} ? ,
              attribute before {xsd:NCName } ? ,
              attribute after {xsd:NCName } ? ,
              attribute min {xsd:normalizedString} ? ,
              element  xtemp:condition { condition_body } *

condition_body =
       attribute key {xsd:normalizedString} ? ,
       attribute lg {xsd:NCName} ? ,
       xsd:normalizedString
```

Semantics:

The <match> element will select one or more event(s) based on a conditional expression specified in the <condition> child element. When a combination of two or more events needs be selected possibly based on different conditional expressions, more than one <match> statement will be specified, and in such case the respective events they match will be distinguished by a symbolic name (match/@event).

- **catch/match/@event**: Gives a symbolic name to the event(s) selected. Usable as a variable later.
- **catch/match/@before**: list by their symbolic names which other event(s) of the same pattern, must occur after this event. These events must be referenced after the @event value(s) of other match statement(s).
- **catch/match/@after**: list which other event(s) of the same pattern, must have occurred prior to this event.
- catch/match/@view: determines how much data from each selected event must be reproduced in the catch result. It is specified as a set of Xpath expressions. If not present, the entire event is returned.
- **catch/match/@min**: determines the minimum number of matching events to select, in order to consider the match as successful. (default = 1)
- **catch/match/condition**: The event selection condition (e.g. XPath expression) that will be evaluated over event instances from the event board .

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

(E1) a purchase order request,

(E2) a PO response (acceptance or rejection),

(E3) a generic receipt to the PO response

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

* Between E1 and E2: the same value for the PO reference# ("order-ref" element here present in the header as a property)

* Between E2 and E3: the MessageID value present in E2, equal to RefMessageID in E3.

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

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

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

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

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

</catch>
```


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

X-effect of a Catch:

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

4.7.5 xtemp:mask

Role: mask an event or a set of events, so that the event(s) is not visible anymore as “lead” event by a catch statement from a scriplet of the same script package.

Compact RelaxNG notation:

```
element xtemp:mask { mask_body }

mask_body =
       attribute events {xsd:normalizedString}
```

Semantics:

The @events attribute contains a sequence (in the XPath sense) of one or more event IDs. This sequence may be obtained from an XPath variable.

Examples:

```
<mask events=”100”/>
```

```
<mask events=”100 101 102”/>
```

In this example, the variable $myevent contains the result of a <catch> statement:

```
<mask events=”{$myevent/xtemp:event/@ID}”/>
```

4.8 The Script Package and its Execution Context

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

4.8.1 xtemp:script-package

Role:

Organize scriplets as a package intended as a unit of deployment and execution. A package also defines referential boundaries for scriplets (a scriplet can only invoke another scriplet in the same package).

Compact RelaxNG notation:

```
element xtemp:script-package { package_body }

package_body =
              attribute name {xsd:NCName} ? ,
              attribute schemaVersionId {xsd:normalizedString} ? ,
              element xtemp:execution-context {excontext_body} ? ,
              (      element xtemp:scriplet {scriplet_body} |
                     element xtemp:function {function_body} ) *
       }
```

4.8.2 xtemp:execution-context

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

Compact RelaxNG notation:

```
element xtemp:execution-context { execontext_body }

execontext_body =
       element xtemp:start-with {
              attribute vptset { xsd:normalizedString } ?,
              attribute scriplet {  xsd:NCName }
              },
       element xtemp:var {var_body} *,
       element xtemp:event-board {board_body} *
```

Semantics:

start-with: Indicates which scriplet among those at top level in the package must be started first

var: Defines a variable global to the script package.

event-board: (see next section)

The execution-context element is optional. When absent, or when start-with is absent, there must be a scriplet named “main” where the execution will start.

4.8.3 xtemp:event-board

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

Role: defines an event board for a script package, that will bind to a physical event source (e.g. an event queue, or a file representing an event log). Such an event board is declared in the execution-context of the script package. Parts of this declaration is processed by the scriplet engine, but the binding to an external event source (or store) and the related conversion into XTemp event format is performed by an event manager function beyond the scope of scriplet processing. Event filters (<xtemp:filter> elements) defined in the declaration provide a "view" over the bound event source, selecting only events from the event source that are of interest for this script package. The event board as used by the script package, is resulting from this binding to a physical event source, followed by a filtering if any.

Compact RelaxNG notation:

```
element xtemp:event-board { evboard_body }

evboard_body =
       attribute name { xsd:NCName } ?,
       attribute event-store { xsd:normalizedString } ?,
       attribute root { xsd:normalizedString } ?,
       attribute mode { ("source" | "sink" | "source-sink") } ?,
       element xtemp:filter {
              attribute name { xsd:normalizedString } ?,
              element xtemp:condition {condition_body} *
```



```
        } * ,
        element xtemp:eventbind {
               attribute id { xsd:normalizedString } ?,
               attribute  timestampexpr { xsd:normalizedString },
               xsd:normalizedString } ?
```

Semantics:

name: the name used for this event board inside this script package.

event-store: a reference to an actual event source to which this event board is to be bound.

mode: defines the mode of use for this event board: as a source of events (or "read-only"), as a sink (or "write-only") or as both.

root: in case of an event board represented as an ad-hoc XML document (i.e. not complying with the event board schema defined as the default for event board structure by this specification) this attribute identifies in the source document used as event board the root container element of which all events are direct children. It must contain an absolute Xpath expression.

filter: Besides these attributes, the event-board element contains zero or more *filters*. In a *source* or *source-sink* mode of use, the filter acts as a selector on the events from the physical event store (after their mapping in XTemp format). The embedded condition - similar in syntax to the match/condition in the <catch> statement - defines an expression (e.g. XPath) used to select events that are qualified for posting on the event board. A condition may define a key (@key) that will be associated with the events it selects. The attribute condition/@key defines an expression which, when evaluated over each selected event, returns the key value associated with this event.

eventbind: this element is used to identify key event attributes used by XTemp when the events are contained in an ad-hoc event board - i.e. are not complying with the event wrapper schema defined as the default by this specification.

The value of the element is an Xpath expression identifying the root (or wrapper) element of each event, relative to the event container element as identified by the event-board/@root expression, if any.

Its attributes are:

- eventbind/@id: (optional) contains an Xpath expression relative to the event wrapper element, that resolves to a string representing a unique sequence number, when applied to an event element. If absent, the position of the XML event element in the event XML container, is used as sequence number.
- eventbind/@timestampexpr: contains an Xpath expression relative to the event wrapper element, that resolves to a date/time usable as timestamp for the event. The date/time must be in standardize format (e.g. 2007-06-15T13:10:13-07:00).

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

Example 1:

```
<!-- binding an event board name to an actual event board -->

<event-board name="PO" event-store="(URI of some physical store)"
mode="source">

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

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

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

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

</event-board>
```

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

Example 2:

```
<!-- binding an event board name to an actual event board -->

<event-board name="PO" event-store="(URI of some physical store)"
mode="source">

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

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

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



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

</event-board>
```

4.9 Message and Event Outputs

Besides X-effects, the other forms of outputs of a script execution are messages and events.

4.9.1 xtemp:message

Role:

Write on the “standard output”, without any format constraint (XML or not).

Compact RelaxNG notation:

```
element xtemp:message {  message_body }
message_body =
       mixed {( eval_body | xeffectitem ) * }
```

4.9.2 xtemp:post

Role:

Posts an event to an event board.

Compact RelaxNG notation:

```
element xtemp:post {
       attribute evboard { xsd:normalizedString } ? ,
       attribute event { xsd:NCName } ? ,
       element xtemp:property { property_body } * ,
       element xtemp:content { xeffectitem }
       }
property_body =
       attribute name { xsd:NCName } ? ,
        xsd:normalizedString
```

Semantics:

The <post> element is posting an event to an event board.

- **post/@evboard**: identifies an event board (logical name). If absent, the posting is produced as part of the X-effect.
- **post/@event**: is a string that contains an event ID that identifies a previously caught event (already posted on an event board). If the event comes from an event board other than the main event board, the logical name of the board must prefix the event ID (e.g. myboard:123)

The content of the element – if present – is event data. When posting, the xtemp processor will add automatically the appropriate event wrapper. For example:

```
<post evboard=”myboard”>
      <xtemp:property name=”source”>monitor123</xtemp:property
      <xtemp:content>
            <abc:mydata/>
      </xtemp:content>
</post>
```

will result in the posting of an event of the form:

```
<xtemp:event id="1" timestamp="2009-06-13T13:12:13-07:00">
      <xtemp:property name=”source”>monitor123</xtemp:property
      <xtemp:content>
            <abc:mydata/>
      </xtemp:content>
</xtemp:event>
```

4.10 Advanced Virtual Present Time Management

4.10.1 xtemp:wait

Role:

Blocks the scriplet execution for some duration. In case of deferred mode of execution, adds the duration to the VP-time.

Compact RelaxNG notation:

```
element xtemp:wait {
      attribute format { xsd:normalizedString } ?,
      attribute until { xsd:normalizedString } ? ,
      attribute for { xsd:normalizedString } ?
      attribute group { xsd:normalizedString } ?
      }
```

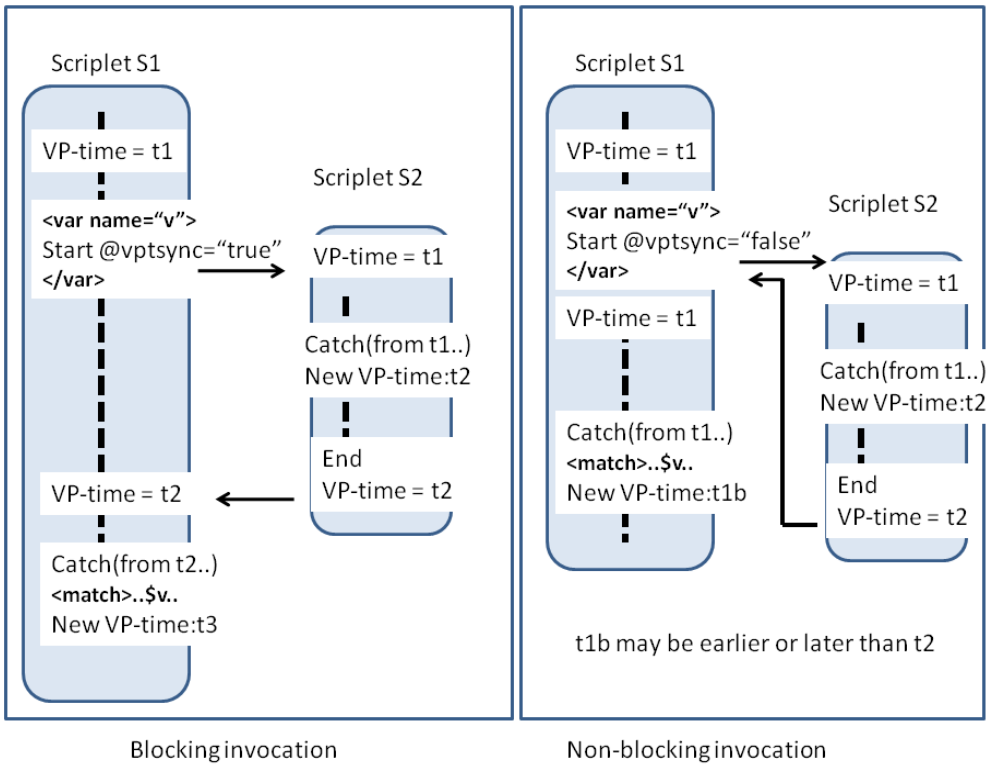
wait: this construct is blocking the scriplet execution for some duration or until a date/time, or until some other scriplets terminate.

- @format: gives the duration or date format to be used, when not complying with the standard format.
- @for: the duration of the waiting, to be added to the current VP-time. In a live execution, this actually means waiting for this duration before resuming execution. This may be an Xpath expression.
- @until: the date until which the wait should be in effect. If @for is also used, the wait will end at the latest of these two dates: (1) vp-time + @for, (2) @until. This may be an Xpath expression.
- @group: The wait operation will wait until all scriplets started within this group have completed. For example, <wait group=”g1”/> will wait for all scriplets started within the group “g1” (as <start scriplet=”X” vptsync=”false” group=”g1”>...) to complete. These scriplets must have been started

within the same scriplet that operates the <wait>, as the group is local to a scriplet. The value of @group may be an expression that evaluates to a group name, if wrapped in "{ }".

4.10.2Managing Scriplet Outputs in a Non-Blocking Mode

When a variable declaration contains a scriplet invocation, this scriplet execution needs to complete before the resulting X-effect can be assigned to the variable. This rule applies regardless of the invocation mode – blocking or not - of the embedded scriplet. The VP-time after the variable declaration will however be affected, according to the rules defined in section 3.2. The following figure is a variant of the figure in 3.2 for blocking vs. non-blocking invocations, showing the use of a variable that captures the result of the invoked scriplet (S2):



In the above figure:

- When this scriplet invocation (here S2) is blocking (start/@vptsync="true") the next statement after this variable declaration will then use a VP-time that corresponds to the end of the invoked scriplet (i.e. its VP-time at the end).
- When this scriplet invocation (here S2) is non-blocking (start/@vptsync="false") the VP-time at the end of the variable execution is not affected by the invocation of the scriplet.

More generally – e.g. in case the variable body contains other statements after the scriplet invocation - the VP-time after the execution of the variable declaration follows the same rules as if the variable declaration were itself a blocking scriplet invocation (see 3.2.2).

In a non-blocking invocation case, the variable ($v in above figure) may be used at a VP-time that is anterior to the VP-time at which it has been set, as illustrated in the above figure: t2 is the VP-time at which the execution of the variable body can be completed (and assigned as its value), but the variable is used at a VP-time (t1) earlier than this time, in S1.

The non-blocking case above illustrates the concurrency semantics of XTemp in terms of its “futures” : the variable V is a “future” the access (or resolution) of which is blocking the invoking scriplet S1 (XTemp has a blocking semantics for its futures).

Consider a modified version of the loop of Example E2 (Appendix A), where the non-blocking invocation of get-PO-response is wrapped in a variable definition ($response_time), so that one can use the resulting response time at each iteration, e.g. for averaging:

```
<loop>
       <var name="po-event">
          <catch>
            <match><condition>xtemp:content/xyz:PO</condition></match>
          </catch>
       </var>
       <until expr="not($po-event/xtemp:event)"/>
       <var name="response_time">
              <start scriplet="get-PO-response" vptsync="false">
          <with-param name="ref" expr="$po-
event/xtemp:event/xtemp:content/xyz:PO/xyz:poref"/>
              </start>
       </var>
       ... (here, use value of $response_time for some averaging)
</loop>
```

Inside the loop, the VP-time after execution of the declaration of the $response_time (future) variable is still the same as before the variable execution, due to the non-blocking invocation of the get-PO-response scriplet. However, the variable execution has “waited” for the end of get-PO-response scriplet to get its output (meaning the get-PO-response scriplet has progressed its own VP-time).

In other words, the variable `$response_time` is a future with blocking semantics for the iteration. However, this blocking does not affect the VP-time of the loop which is only affected by the event catching inside `$po-event`.

As a result, when performing the next loop iteration, the catching of the next PO event (as captured by the $po-event variable) may be done at a VP-time anterior to the VP-time at which the response to the previous PO event has been caught. This is the desired semantics, as these PO transactions can be interleaved in the event board (i.e. executed with some overlap).

This examples shows that it is impossible to execute some XTemp scripts "live" i.e. consuming events from an input event board in the temporal order of their occurrence: such script must at some point capture events (in “future” variables) before catching an earlier event (like the above example). Note that this is not contradictory with the rule requiring the VP-time in a scriplet to always monotonically increase as a function of the actual execution time: the catching of “future” events when done by another non-blocking scriplet, will not affect the VP-time of the invoking scriplet This is only possible with the availability of an event board to give access to past events and a flexible management of virtual time.

4.10.3Joining Concurrent Scriplets

It may be useful for a scriplet to start some scriplet(s) in a non-blocking way, yet to know about the end VP-time of these scriplets e.g. in order to wait for their completion before executing some operation.

One way for an invoking scriplet S1 to obtain the ending VP-time of an invoked non-blocking scriplet S2, is to invoke S2 inside a variable definition, while generating some X-effect at the end of S2 that will display the VP-time, e.g. <myreport:endvpt><xtemp:eval expr=”$currentvpt”/></myreport:endvpt> . Then S1 can extract this value from the variable that invoked S2.

However, when the objective is to join non-blocking scriplets, a simpler way is to use the start/@group attribute. All scriplets to be joined must have been started in the same group - e.g. “g1” :

<start scriplet=”S2” group=”g1” vptsync=”false”>...

Then the invoking scriplet may wait for all the scriplet for this group to be complete by stating:

<wait group=”g1”/>

4.11 Functions

Functions allow for either one or both of the following benefits:

- advanced computations that may exceed the capability of XTemp scripting.
- Invocation from inside expressions (e.g. inside an Xpath expression used anywhere Xtemp expressions are allowed.)

Functions may be written in Xtemp or in "foreign" languages or scripts, e.g. in Java, C# or XSLT, making it possible to reuse advanced function libraries and operators available for these languages, while scriplets are exclusively coded in XTemp syntax.

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

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

```
<function name="purchaseorder">
    <param name="POref"/>
    <param name="cust"/>
    <param name="amount"/>
    <var name="v1">
          <myapp:PO>
        <POReference><eval expr="POref"/></POReference>
        <Customer><eval expr="$cust"/></Customer>
        <Amount><eval expr="$amount"/></Amount>
      </myapp:PO>
        </var>
             <eval expr="$v1"/>
  </function>
```

When written in Xtemp, the X-effect of the body of a function is used as return value as above.

The following function is calculating (recursively) the factorial of a number:

```
 <function name="xyz:factorial">
    <param name="num"/>
    <decide>
      <if expr="$num eq 1">1</if>
      <else>
        <eval expr="$num * xyz:factorial($num - 1)"/>
      </else>
    </decide>
  </function>
```

It may be used inside Xpath expressions. In the folowing examples, both Xpath expressions would select the 6th lineitem sub-element:

```
    <eval expr="$myelement//abc:lineitem[xyz:factorial(3)]"/>
    <eval expr="$myelement//abc:lineitem[xyz:factorial(4) div 4]/@name"/
```


Compact RelaxNG notation:

```
element xtemp:function {function_body }

function_body =
       attribute name { xsd:normalizedString },
       element xtemp:param { param_body } *,
         ( fnxtemptatement | fnxeffectitem ) *

fnxeffectitem =
  element *:* - xtemp:* { ( fnxtemptatement | fnxeffectitem )* }

fnxtemptatement =
  (
       element xtemp:var { infn_var_body } |
       element xtemp:decide { infn _decide_body } |
       element xtemp:eval { infn_eval_body } |
       element xtemp:exit { infn_exit_body } |
       element xtemp:if { infn_if_body } |
       element xtemp:message { message_body } |
  )
```

NOTE: the types denoted "infn_..." are similar to previous types for these various XTemp statements, except they do not allow nesting of statements other than those listed in the xtfuncstatement type.

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

In contrast, a scriplet does not explicitly return a result. However, its execution produces an X-effect - analogous to a side-effect - which is the concatenation of the X-effect of each step inside it.

4.12 Adapters

4.12.1 xtemp:call-adapter

Role:

Invoke an external adapter or program.

A particular class of adapters are **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 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 scriplet.

Compact RelaxNG notation:

```
element xtemp:call-adapter {
       attribute name { xsd:NCName } ,
       attribute type { xsd:normalizedString } ?
       element  xtemp:with-param { withparam_body } *
       }
```


Semantics:

The <call-adapter> element is invoking an adapter, which will have the effect of posting some event on an event board. This event may come from an external source (mediated by the adapter) or may be created by the adapter itself from data passed as parameters (in which case the <call-adapter> statement has the same semantics as a <post>).

5 Event Model and Management

5.1 General Rules

XTemp does not require a particular event structure – but events must follow a few minimal rules (see below). The event XML schema presented here (within the XTemp namespace) and in Appendix B is optional. However it will be the default schema in case no other is assumed by the XTemp script (e.g. if no xtemp:event-board/xtemp:eventbind element is specified in the script.)

In all cases, the following rules are assumed to be complied with, and represent the minimal assumptions:

Rule 1: Events have an XML representation. This representation has a single element at its root. It may contain non-XML data, but each part of an event that is to be queried or accessed in XTemp statements, must be accessible with a single Xpath expression from the root element.

Rule 2: Each event has a "**timestamp**" - a date/time attribute that represents the occurrence time of the event . This attribute must be identified and accessed in the same way for all events of a same type in the same event-board. The timestamp is considered as the time at which the event occurred from an XTemp processing viewpoint and is the only date/time that is taken into account by the XTemp engine, when catching events occurring after a given VP-time. The timestamp may be different from the event creation time or "business time" as set by the event originator – e.g. the business time is a purchase order creation time, while the timestamp may be the time at which the PO message was actually captured and posted to the event board by the monitoring system. Which one of the possibly multiple date/time items associated with an event should be used as timestamp is a configuration choice. In case the business time is not used as timestamp, it is still accessible to scriplets as any other event data item and may be used in event selection expressions – it is simply not the date/time the XTemp engine will refer to when comparing the event time to VP-time or when setting the next VP-time value.

Rule 3: An event board must order events based on the timestamp, and facilitate selective, ordered access to events occurring within any time window – past, current or future – i.e. events occurring between two date/time values. The event board must also allow for queue-like access from several scriplets concurrently, using the VP-time assigned for each scriplet.

The rest of this section is describing a RECOMMENDED default structure for events.

5.2 Recommended Event Structure

Any event and event board that complies with the three above rules listed in 4.1, can be processed by XTemp. This section describes a RECOMMENDED event structure that is open to diverse types of event contents: events can be as diverse as

- (a) monitoring reports about an indicator's value over time (e.g. snapshots of business indicators or quotes),
- (b) traces of B2B message exchanges including business documents,
- (c) alarms and error reports,
- (d) contextual information about other events, such as system configuration information, or meta-data document (contract, interface description, policy in force, etc.)

Consequently, the event model is here mostly defining an event wrapper. This event wrapper allows for distinguishing different "event types". Such types are not built-in but defined by users in order to match categories of events in use for some particular monitoring application, or generated by specific event sources.

The event wrapper distinguishes three layers in an event data:

- Header attributes
- Properties
- Content or Payloads

5.2.1 Header Attributes

The following represents RECOMMENDED attributes for XTemp events. Such attributes are playing an important role in event management or event correlation. These attributes are represented as attributes of the XML element <event>. Example:

<xtemp:event id="1" timestamp="2009-10-13T13:12:13-07:00">

Some of these attributes are predefined and their support is built-in the XTemp processor (see below). Others can be defined when defining various event types associated with an application or an event board.

The built-in header attributes are:

- **id**: an integer value which is a sequence number assigned to the event at the time of its posting to the EB. This value is automatically assigned by the event manager. The @id uniquely identifies the event in a particular event board. The @id defines a total order for the events in the board.
- **timestamp**: date/time value representing the event occurring time (timestamp). This is the date/time taken into account by the XTemp engine, when catching events occurring after a given VP-time. The @timestamp value may be automatically assigned by the event board manager when the event is posted, or may be provided by the event originator. In case the @timestamp attribute value is not provided when posting an event, and if a configuration statement exists in the event board that associates an XPath expression with this type of event, for accessing a date/time field, then this date/time value will be used as timestamp value. The timestamp defines a partial order over the events of an event board, because two events could have the same timestamp value. However, the timestamp ordering is consistent with the ID ordering: for any two events E1 and E2, timestamp (E1) < timestamp (E2) => ID(E1) < ID(E2).
- **type**: (optional) a string value identifying the event type. Each @type value is associated with a fixed set of header properties (list of "names" in the list of name-value pairs). Each @type value is also associated with a same way to identify the “timestamp” of the event (i.e. as an internal data field) in case the @timestamp attribute value is not provided by the event originator.

5.2.2 Header Properties

Each property item is a name-value pair.

Each property is represented by the XML element <property>. Example:

<xtemp:property name=”price">200</ xtemp:property>

The list of property names may vary from one event to the other, but is specific to an event type. The value of each property is typed: it is either an atomic value of a conventional datat type, or an XML fragment. Properties may be selected items from the content of the original event (i.e. from event

payload), and represent an easy-to-access abstraction of it. They may contain references to external objects not managed by the Event Board.

5.2.3 Event Payload(s)

One or more payloads can be associated with an event. A payload is wrapped using the <content> element. The payload may be either wrapped itself as part of the event, or is an external resource referenced by the event.

Example:

```
<xtemp:event id="1" timestamp="2009-10-13T13:12:13-07:00">
  <xtemp:content>
    <myapp:PurchaseOrder>
      <myapp:poref>Name1</myapp:poref>
    </myapp:PurchaseOrder>
  </xtemp:content>
</xtemp:event>
```

The event wrapper schema is provided in Appendix B.

5.3 Event Boards

5.3.1 Using an Event Board

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

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

(a) a read-only event board representing a log of events to be analyzed,

(b) a temporary event board for the sake of internal synchronization of the script execution, i.e. for which the script has read-write access,

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

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

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

5.3.2 Event Visibility Window

The "event visibility window" of a scriplet over an event board determines the set of events that are accessible by the scriplet at some given time. The same event board may have different visibility windows for the different scriplets that access it. Generally there is then one visibility window for each pair (Event Board, Scriplet).

The visibility window of a scriplet for a particular event board, is defined by:

- the Virtual Present time (VP-time), which is specific to the execution of the scriplet instance and evolves during the execution.
- masked events, which makes events invisible to the scriplet instance.

These operations affect the visibility windows of an event board:

- **Catch** and **Wait** These operations modify in general the VP-time of an invoking scriplet.
- **Post**: This operation may add new events to the event board, thus modifying the set of events visible to the invoking scriplet or to another scriplet.
- **Mask**: Event masking: once a <catch> statement has captured some event(s), it may be necessary to mask these events so that they will not be caught again by subsequent <catch> operations. This modifies the set of accessible events for the remaining execution of the scriplet, or of any scriplet inside the same script package.

5.3.3 Event Adapters

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

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

6 Conformance

Two levels of conformance are defined:

- Core XTemp conformance level
- Full XTemp conformance level

6.1 Core Conformance Level

The core level of conformance is based on a subset of the XTemp language features as defined in Section 4, and also reduces the options for several of these features. This subset of the language is implementable with XSLT2.0.

6.1.1 Features not required to be supported

These statements are not required to be supported in the “Core” Xtemp conformance level:

- post
- call-adapter
- function
- mask
- decide
- wait

The reserved variable @currentvpt is not required to be implemented.

6.1.2 Features for which restricted support is allowed

execution-context:

- the element <event-board> is not required to be supported. Scripts then use a single “default” un-named event board as input, and use the default event model.

start:

- the @bubble-exit and @group attributes are not required to be supported (meaning the default value “false” is assumed).

catch:

- Attributes @eboard, and @vptset are not required to be supported.
- Only one <match> statement is required to be supported as child element, and none of its attributes is required to be supported.

exit:

- none of its attributes is required to be supported.

6.2 Full Conformance Level

The full level of conformance requires support for all the XTemp language features as defined in Section 4.

Appendix A. Examples

A.1. Complete Example E1

This example illustrates the following features:
- use of variables and their types.
- use of parameterized in-line X-effects.
- use of the xtemp:eval statement.
Some noteworthy aspects of this examples are:
- The example is not processing any event (no event board is declared).
- The variable "mynewpo" is assigned an XML fragment that is parameterized by XTemp statements evaluated at the time of assignment; xtemp:eval, xtemp:decide. At the time this variable is passed as parameter to the scriplet "addpo", it is a fully instantiated XML fragment itself assigned to the parameter "mypo". The value of this parameter is then produced as part of the final output X-effect, wrapped in the <report:latestpo> element.

```
<script-package xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" schema-
version-id="0.4"
             xsi:schemaLocation="http://docs.oasis-open.org/tamie/xtemp/200909
http://www.oasis-open.org/committees/download.php/34985/xtemp-0-6-3.xsd"
             xmlns="http://docs.oasis-open.org/tamie/xtemp/200909"
             xmlns:myapp="http://www.abc.com/TBD/myapp"
             xmlns:report="http://www.abc.com/TBD/report"
             name="CUC-1">

  <execution-context>
    <!-- starting point -->
    <start-with scriplet="test1"/>
    <!-- a global variable -->
    <var name="currentTotal" type="double">1000</var>
  </execution-context>

  <!-- ======= scriplet: test1 ======= -->
  <scriplet name="test1">

    <var name="totalpos" type="double" expr="$currentTotal"/>
    <var name="minimum" type="integer">1</var>
    <var name="mynewpo" type="xml">
      <myapp:po>
        <myapp:ref>123456</myapp:ref>
        <myapp:product>greenbike</myapp:product>
        <myapp:qty><eval expr="$minimum + 1"/></myapp:qty>
        <decide>
                <if expr="$minimum + 1 gt 10">
            <myapp:price>130</myapp:price>
          </if>
              <else>
            <myapp:price>165</myapp:price>
              </else>
        </decide>
      </myapp:po>
    </var>
    <var name="mydiscountrate" type="integer">8</var>

    <report:totalUpdate>

      <start scriplet="addpo">
```

```
        <with-param name="mypo" expr="$mynewpo"/>
        <with-param name="previoustotal" expr="$totalpos"/>
        <with-param name="mydiscountrate" expr="$mydiscountrate"/>
      </start>

    </report:totalUpdate>

  </scriplet> <!-- test1 -->

  <!-- ======= scriplet: addpo ======= -->
  <scriplet name="addpo">
    <param name="mypo"/>
    <param name="previoustotal"/>
    <param name="mydiscountrate"/>

    <report:latestpo><eval expr="$mypo"/></report:latestpo>
    <report:previous><eval expr="$previoustotal"/></report:previous>
    <report:new><eval expr="$previoustotal + ($mypo/myapp:po/myapp:price *
$mypo/myapp:po/myapp:qty) * (100 - $mydiscountrate) div 100 "/></report:new>

  </scriplet> <!-- addpo -->

</script-package>
```

The X-effect resulting from the execution of this script, is a test report about the update of a "total" amount (previously set at $1000) based on an incoming purchase order (stored in a variable in the previous script). The update is represented by the report:totalUpdate element, which contains three children elements:
(a) an XML fragment representing the purchase order,
(b) previous amount on the customer balance (report:previous element)
(c) new amount on the customer balance (report:new element)

```
<report:totalUpdate
   xmlns:myapp="http://www.abc.com/TBD/myapp"
   xmlns:report="http://www.abc.com/TBD/report"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xtemp="http://docs.oasis-open.org/tamie/xtemp/200909"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://docs.oasis-open.org/tamie/xtemp/200909">
  <report:latestpo>
    <myapp:po>
      <myapp:ref>123456</myapp:ref>
      <myapp:product>greenbike</myapp:product>
      <myapp:qty>2</myapp:qty>
      <myapp:price>165</myapp:price>
    </myapp:po>
  </report:latestpo>
  <report:previous>1000</report:previous>
  <report:new>1303.6</report:new>
</report:totalUpdate>
```

A.2. Complete Example E2

This example illustrates:
- use of loops
- event catching
- non-blocking starting of scriplets

This example illustrates the convenience of concurrent (non-blocking) scripting, even when executing over logged event sources (e.g. just for analyzing past event logs). In the "main" scriplet, a loop is catching the lead event of each purchase order (PO) transaction. For each PO, the loop is starting a scriplet

("get-PO-response") that will follow this particular transaction. By being started non-blocking, the "get-PO-response" scriplet can track each PO transaction using its own VP-time starting at PO reception, and its execution logic is not affected by the occurrence of other concurrent PO transactions.
Regardless of the invocation mode (blocking or non-blocking) of the invoked "get-PO-response" scriplet, the X-effect of each one of its instances will be serialized following the iteration order (i.e. in the order of reception of each PO), in the final monitoring report.
The object of this script is to monitor each PO duration, and report whether the entire PO transaction (PO request + PO confirmation) runs within 10mn or not. Note that in case the transaction lasts more than 10mn, we are not interested here in knowing exactly how long it runs: the <catch> element is using then a 600 sec timeout (@tryfor).

```
<script-package xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://docs.oasis-open.org/tamie/xtemp/200909
http://www.oasis-open.org/committees/download.php/34985/xtemp-0-6-3.xsd"
             xmlns="http://docs.oasis-open.org/tamie/xtemp/200909"
             xmlns:abc="abcabc"
             xmlns:myns="myns"
             xmlns:xyz="xyz">

  <scriplet name="main">
    <abc:results>
      <loop>
       <var name="po-event">
          <catch>
            <match><condition>xtemp:content/xyz:PO</condition></match>
          </catch>
       </var>
       <until expr="not($po-event/xtemp:event)"/>
       <start scriplet="get-PO-response" vptsync="false">
          <with-param name="ref" expr="$po-
event/xtemp:event/xtemp:content/xyz:PO/xyz:poref"/>
       </start>
      </loop>
    </abc:results>
  </scriplet>

  <scriplet name="get-PO-response">
    <param name="ref" type="string"/>

    <var name="confirm-event">
      <catch tryfor="PT600S">
        <match><condition>xtemp:content/xyz:confirm[xyz:poref =
$ref]</condition></match>
      </catch>
    </var>
    <if expr="not($confirm-event/xtemp:event)">
      <abc:messg>No confirm event for <eval expr="$ref"/> received within 10
minutes of the PO</abc:messg>
      <exit/>
    </if>
    <abc:messg>Confirmation received for PO <eval expr="$ref"/></abc:messg>
  </scriplet>
</script-package>
```

The event log contains the following set of events, showing three PO transactions overlapping in time:

```
<xtemp:event-board xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xtemp="http://docs.oasis-open.org/tamie/xtemp/200909"
 xmlns:xyz="xyz"
 xsi:schemaLocation="http://docs.oasis-open.org/tamie/xtemp/200909
http://www.oasis-open.org/committees/download.php/34094/xtemp-0-4.xsd"
 timestamp="2007-06-13T13:10:03-07:00" schema-version-id="0.4">
  <xtemp:events>
    <xtemp:event id="1" timestamp="2007-06-13T13:10:13-07:00">
```



```
      <xtemp:content>
        <xyz:PO>
          <xyz:poref>Name1</xyz:poref>
        </xyz:PO>
      </xtemp:content>
    </xtemp:event>

    <xtemp:event id="2" timestamp="2007-06-13T13:12:13-07:00">
      <xtemp:content>
        <xyz:PO>
          <xyz:poref>Name2</xyz:poref>
        </xyz:PO>
      </xtemp:content>
    </xtemp:event>

    <xtemp:event id="3" timestamp="2007-06-13T13:13:13-07:00">
      <xtemp:content>
        <xyz:PO>
          <xyz:poref>Name3</xyz:poref>
        </xyz:PO>
      </xtemp:content>
    </xtemp:event>

    <xtemp:event id="4" timestamp="2007-06-13T13:15:53-07:00">
      <xtemp:content>
        <xyz:confirm>
          <xyz:poref>Name2</xyz:poref>
        </xyz:confirm>
      </xtemp:content>
    </xtemp:event>

    <xtemp:event id="5" timestamp="2007-06-13T13:18:53-07:00">
      <xtemp:content>
        <xyz:confirm>
          <xyz:poref>Name1</xyz:poref>
        </xyz:confirm>
      </xtemp:content>
    </xtemp:event>

    <xtemp:event id="6" timestamp="2007-06-13T13:33:53-07:00">
      <xtemp:content>
        <xyz:confirm>
          <xyz:poref>Name3</xyz:poref>
        </xyz:confirm>
      </xtemp:content>
    </xtemp:event>
  </xtemp:events>
</xtemp:event-board>
```

The resulting X-effect of running the script over this event log is as below. The X-effect for each PO is an <abc:messg> element that reports if the transaction took less than 10mn or more.

```
<abc:results
            xmlns:translator="translator"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:xtemp="http://docs.oasis-open.org/tamie/xtemp/200909"
            xmlns:abc="abcabc" xmlns="http://docs.oasis-
open.org/tamie/xtemp/200909" >
  <abc:messg>Confirmation received for PO Name1</abc:messg>
  <abc:messg>Confirmation received for PO Name2</abc:messg>
  <abc:messg>No confirm event for Name3 received within 10 minutes of the
PO</abc:messg>
</abc:results>
```


Appendix B. Event Wrapper Schema

The default event wrapper schema is :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
  targetNamespace="http://docs.oasis-open.org/tamie/xtemp/200909"
xmlns="http://docs.oasis-open.org/tamie/xtemp/200909" version="0.4">
  <xs:element name="event-board" type="event-board_type"/>
  <xs:element name="event" type="event_type"/>
  <xs:complexType name="content_type" mixed="true">
    <xs:sequence>
      <xs:any namespace="##any" processContents="skip" maxOccurs="unbounded"
minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:normalizedString"/>
    <xs:attribute name="schema" type="xs:normalizedString"/>
  </xs:complexType>
  <xs:complexType name="event-board_type">
    <xs:sequence>
      <xs:element name="events" type="events_type"/>
    </xs:sequence>
    <xs:attribute name="schema-version-id" type="xs:normalizedString"/>
    <xs:attribute name="id" type="xs:normalizedString"/>
    <xs:attribute name="timestamp" type="xs:dateTime"/>
  </xs:complexType>
  <xs:complexType name="event_type">
    <xs:sequence>
      <xs:element name="content" type="content_type" minOccurs="0"/>
      <xs:element name="event-properties" type="event-properties_type"
minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:normalizedString"/>
    <xs:attribute name="timestamp" type="xs:dateTime"/>
    <xs:attribute name="origin-time" type="xs:dateTime"/>
    <xs:attribute name="event-type" type="xs:normalizedString"/>
  </xs:complexType>
  <xs:complexType name="events_type">
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="event"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="event-properties_type">
    <xs:sequence>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="property"
type="property_type"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="property_type" mixed="true">
    <xs:sequence>
      <xs:any namespace="##any" processContents="skip" maxOccurs="unbounded"
minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:normalizedString"/>
  </xs:complexType>
</xs:schema>
```

Appendix C. XTemp markup Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://docs.oasis-open.org/tamie/xtemp/200909"
  elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://docs.oasis-open.org/tamie/xtemp/200909">
  <!-- XTemp -->
  <!-- Top level global element -->
  <xs:element name="script-package" type="script-package_type"/>
  <!-- Global elements -->
  <xs:element name="call-adapter" type="call-adapter_type"/>
  <xs:element name="catch" type="catch_type"/>
  <xs:element name="decide" type="decide_type"/>
  <xs:element name="eval" type="eval_type"/>
  <xs:element name="exit" type="exit_type"/>
  <xs:element name="function" type="function_type"/>
  <xs:element name="if" type="if_type"/>
  <xs:element name="loop" type="loop_type"/>
  <xs:element name="mask" type="mask_type"/>
  <xs:element name="message" type="message_type"/>
  <xs:element name="post" type="post_type"/>
  <xs:element name="scriplet" type="scriplet_type"/>
  <xs:element name="start" type="start_type"/>
  <xs:element name="var" type="var_type"/>
  <xs:element name="wait" type="wait_type"/>
  <!-- Global types -->
  <xs:complexType name="call-adapter_type">
    <xs:sequence>
      <xs:element name="with-param" minOccurs="0" maxOccurs="unbounded"
type="withparam_type"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:NCName" use="required"/>
    <xs:attribute name="type" type="xs:normalizedString" use="optional"/>
  </xs:complexType>
  <xs:complexType name="catch_type">
    <xs:sequence>
      <xs:element name="match" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="condition" type="condition_type" minOccurs="0"/>
          </xs:sequence>
          <xs:attribute name="event" type="xs:NCName" use="optional"/>
          <xs:attribute name="min" type="xs:normalizedString" use="optional"/>
          <xs:attribute name="before" type="xs:NCName" use="optional"/>
          <xs:attribute name="after" type="xs:NCName" use="optional"/>
          <xs:attribute name="view" type="xs:normalizedString"
use="optional"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="tryfor" type="xs:normalizedString" use="optional"/>
    <xs:attribute name="eboard" type="xs:NCName" use="optional"/>
    <xs:attribute name="vptset" type="xs:normalizedString" use="optional"/>
    <xs:attribute name="vptend" type="xs:normalizedString" use="optional"/>
  </xs:complexType>
  <xs:complexType name="condition_type">
    <xs:simpleContent>
      <xs:extension base="xs:normalizedString">
        <xs:attribute name="key" type="xs:normalizedString" use="optional"/>
        <xs:attribute name="lg" type="xs:NCName"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
```

```
  <xs:complexType name="decide_type">
    <xs:sequence>
      <xs:element name="if" type="if_type"/>
      <xs:element name="else-if" type="if_type" minOccurs="0"
maxOccurs="unbounded"/>
      <xs:element name="else" type="if-else_type"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="decide_function_type">
    <xs:sequence>
      <xs:element name="if" type="if_function_type"/>
      <xs:element name="else-if" type="if_function_type" minOccurs="0"
maxOccurs="unbounded"/>
      <xs:element name="else" type="if-else_function_type"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="eval_type">
    <xs:attribute name="expr" type="xs:normalizedString" use="required"/>
    <xs:attribute name="lg" type="xs:NCName" use="optional"/>
  </xs:complexType>
  <xs:complexType name="eval_function_type">
    <xs:attribute name="expr" type="xs:normalizedString" use="required"/>
    <xs:attribute name="lg" type="xs:NCName" use="optional"/>
  </xs:complexType>
  <xs:complexType name="execution-context_type">
    <xs:sequence>
      <xs:element name="start-with">
        <xs:complexType>
          <xs:attribute name="vptset" type="xs:normalizedString"
use="optional"/>
          <xs:attribute name="scriplet" type="xs:NCName" use="required"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="var" minOccurs="0" maxOccurs="unbounded"
type="var_type"/>
      <xs:element name="event-board" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="filter" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="condition" type="condition_type"
minOccurs="0"/>
                </xs:sequence>
                <xs:attribute name="name" type="xs:normalizedString"
use="optional"/>
              </xs:complexType>
            </xs:element>
            <xs:element name="eventbind" minOccurs="0">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:normalizedString">
                    <xs:attribute name="id" type="xs:normalizedString"
use="optional"/>
                    <xs:attribute name="timestampexpr"
type="xs:normalizedString" use="required"/>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
          <xs:attribute name="name" type="xs:NCName" use="optional"/>
          <xs:attribute name="event-store" type="xs:normalizedString"
use="optional"/>
```



```
          <xs:attribute name="root" type="xs:normalizedString"
use="optional"/>
          <xs:attribute name="mode" use="optional">
            <xs:simpleType>
              <xs:restriction base="xs:normalizedString">
                <xs:enumeration value="source"/>
                <xs:enumeration value="sink"/>
                <xs:enumeration value="source-sink"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="exit_function_type">
    <xs:group ref="xeffectitem_group" minOccurs="0" maxOccurs="unbounded"/>
    <xs:attribute name="message" type="xs:normalizedString" use="optional"/>
    <xs:attribute name="result" type="xs:normalizedString" use="optional"/>
  </xs:complexType>
  <xs:complexType name="function_type">
    <xs:sequence>
      <xs:element name="param" minOccurs="0" maxOccurs="unbounded"
type="param_type"/>
      <xs:group ref="functiongeneral_group" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:normalizedString" use="required"/>
  </xs:complexType>
  <xs:complexType name="if_type">
    <xs:complexContent>
      <xs:extension base="if-else_type">
        <xs:attribute name="expr" type="xs:normalizedString" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="if_function_type">
    <xs:complexContent>
      <xs:extension base="if-else_function_type">
        <xs:attribute name="expr" type="xs:normalizedString" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="if-else_type">
    <xs:group ref="general_group" minOccurs="0" maxOccurs="unbounded"/>
  </xs:complexType>
  <xs:complexType name="if-else_function_type">
    <xs:group ref="functiongeneral_group" minOccurs="0"
maxOccurs="unbounded"/>
  </xs:complexType>
  <xs:complexType name="loop_type">
    <xs:sequence>
      <xs:element name="on-start" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="lvar" minOccurs="0" maxOccurs="unbounded"
type="var_type"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:group ref="loop_group" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="lvar-next" minOccurs="0" maxOccurs="unbounded"
type="var_type"/>
      <xs:element name="on-final" minOccurs="0">
        <xs:complexType>
```



```
          <xs:group ref="general_group" minOccurs="0" maxOccurs="unbounded"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="mask_type">
    <xs:attribute name="events" type="xs:normalizedString" use="required"/>
  </xs:complexType>
  <xs:complexType name="message_type" mixed="true">
    <xs:group ref="message-group" minOccurs="0" maxOccurs="unbounded"/>
  </xs:complexType>
  <xs:complexType name="param_type" mixed="true">
    <xs:attribute name="name" type="xs:NCName" use="required"/>
    <xs:attribute name="type" type="xs:NCName" use="optional"/>
  </xs:complexType>
  <xs:complexType name="post_type">
    <xs:sequence>
      <xs:element name="property" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="xs:normalizedString">
              <xs:attribute name="name" type="xs:NCName" use="optional"/>
            </xs:extension>
          </xs:simpleContent>
        </xs:complexType>
      </xs:element>
      <xs:element name="content" minOccurs="1" maxOccurs="1">
        <xs:complexType>
          <xs:group ref="xeffectitem_group" minOccurs="0"
maxOccurs="unbounded"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="event" type="xs:NCName" use="optional"/>
    <xs:attribute name="evboard" type="xs:normalizedString" use="optional"/>
  </xs:complexType>
  <xs:complexType name="scriplet_type">
    <xs:sequence>
      <xs:element name="param" minOccurs="0" maxOccurs="unbounded"
type="param_type"/>
      <xs:group ref="general_group" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:NCName" use="required"/>
  </xs:complexType>
  <xs:complexType name="script-package_type">
    <xs:sequence>
      <xs:element name="execution-context" type="execution-context_type"
minOccurs="0"/>
      <xs:group ref="script-package_group" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:NCName" use="optional"/>
    <xs:attribute name="schemaVersionId" type="xs:normalizedString"
use="optional"/>
  </xs:complexType>
  <xs:complexType name="start_type">
    <xs:sequence>
      <xs:element name="with-param" minOccurs="0" maxOccurs="unbounded"
type="withparam_type"/>
    </xs:sequence>
    <xs:attribute name="bubble-exit" type="xs:boolean" use="optional"/>
    <xs:attribute name="scriplet" type="xs:NCName" use="required"/>
    <xs:attribute name="vptset" type="xs:normalizedString" use="optional"/>
    <xs:attribute name="vptsync" type="xs:boolean" use="optional"/>
    <xs:attribute name="group" type="xs:NCName" use="optional"/>
```



```
  </xs:complexType>
  <xs:complexType name="var_type" mixed="true">
    <xs:group ref="general_group" minOccurs="0" maxOccurs="unbounded"/>
    <xs:attribute name="name" type="xs:NCName" use="required"/>
    <xs:attribute name="type" type="xs:NCName" use="optional"/>
    <xs:attribute name="expr" type="xs:normalizedString" use="optional"/>
  </xs:complexType>
  <xs:complexType name="var_function_type" mixed="true">
    <xs:group ref="functiongeneral_group" minOccurs="0"
maxOccurs="unbounded"/>
    <xs:attribute name="name" type="xs:NCName" use="required"/>
    <xs:attribute name="type" type="xs:NCName" use="optional"/>
    <xs:attribute name="expr" type="xs:normalizedString" use="optional"/>
  </xs:complexType>
  <xs:complexType name="wait_type">
    <xs:attribute name="format" type="xs:normalizedString" use="optional"/>
    <xs:attribute name="until" type="xs:normalizedString" use="optional"/>
    <xs:attribute name="for" type="xs:normalizedString" use="optional"/>
    <xs:attribute name="group" type="xs:normalizedString" use="optional"/>
  </xs:complexType>
  <xs:complexType name="withparam_type" mixed="true">
    <xs:group ref="general_group" minOccurs="0" maxOccurs="unbounded"/>
    <xs:attribute name="expr" type="xs:normalizedString" use="optional"/>
    <xs:attribute name="name" type="xs:NCName" use="required"/>
  </xs:complexType>
  <!-- groups -->
  <xs:group name="general_group">
    <xs:choice>
      <xs:group ref="xtempstatement_group"/>
      <xs:group ref="xeffectitem_group"/>
    </xs:choice>
  </xs:group>
  <xs:group name="functiongeneral_group">
    <xs:choice>
      <xs:group ref="xtfuncstatement_group"/>
      <xs:group ref="fnxeffectitem_group"/>
    </xs:choice>
  </xs:group>
  <xs:group name="xtempstatement_group">
    <xs:choice>
      <xs:element ref="mask"/>
      <xs:element ref="call-adapter"/>
      <xs:element ref="catch"/>
      <xs:element ref="decide"/>
      <xs:element ref="eval"/>
      <xs:element ref="exit"/>
      <xs:element ref="if"/>
      <xs:element ref="loop"/>
      <xs:element ref="message"/>
      <xs:element ref="post"/>
      <xs:element ref="start"/>
      <xs:element ref="var"/>
      <xs:element ref="wait"/>
    </xs:choice>
  </xs:group>
  <xs:group name="xtfuncstatement_group">
    <xs:choice>
      <xs:element name="var" type="var_function_type"/>
      <xs:element name="decide" type="decide_function_type"/>
      <xs:element name="eval" type="eval_function_type"/>
      <xs:element name="exit" type="exit_function_type"/>
      <xs:element name="if" type="if_function_type"/>
      <xs:element ref="message"/>
    </xs:choice>
  </xs:group>
```



```
  <xs:group name="xeffectitem_group">
    <xs:sequence>
      <xs:any namespace="##other" processContents="lax"/>
    </xs:sequence>
  </xs:group>
  <xs:group name="fnxeffectitem_group">
    <xs:sequence>
      <xs:any namespace="##other" processContents="lax"/>
    </xs:sequence>
  </xs:group>
  <xs:group name="loop_group">
    <xs:choice>
      <xs:group ref="general_group"/>
      <xs:element name="until">
        <xs:complexType>
          <xs:attribute name="expr" type="xs:normalizedString"
use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:group>
  <xs:group name="message-group">
    <xs:choice>
      <xs:element ref="eval"/>
      <xs:group ref="xeffectitem_group"/>
    </xs:choice>
  </xs:group>
  <xs:group name="script-package_group">
    <xs:choice>
      <xs:element ref="function"/>
      <xs:element ref="scriplet"/>
    </xs:choice>
  </xs:group>
  <xs:complexType name="exit_type" mixed="true">
    <xs:group ref="message-group" minOccurs="0" maxOccurs="unbounded"/>
    <xs:attribute name="message" type="xs:normalizedString" use="optional"/>
    <xs:attribute name="result" type="xs:normalizedString" use="optional"/>
  </xs:complexType>
</xs:schema>
```

Appendix D. Acknowledgments

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

Participants:

- Stephen Green
- Hyunbo Cho
- Dale Moberg
- Chuck Morris
- Jacques Durand

Appendix E. Revision History

Rev	Date	By Whom	What
WD v0.87	12/23/10	Jacques Durand	Initial complete draft
WD v0.9	01/20/11	Jacques Durand	**Extensive editorial changes in these sections**: Section 2: reworded 2.1, 2.3 Section 3: reworded 3.1, extended and reworded 3.3 (new figures), new subsection 3.12 Section 4: reworded 4.1 (major design change to accommodate non-xtemp events, also in 3.10.3) **General design changes**: - refined the semantics of VP-time: rename start/@concurrent as start/@vptsyn, added a @currentvpt reserved variable, - enhanced the <wait> semantics to be able to join scriplet threads (3.12.1), added a notion of “group” for joining such scriplet executions (see 3.12.3, and new start/@group attribute.) - extension of the execution-context element with new element “eventbind” (3.10.3).
WD v0.91			Added update proposals following-up public comments (from M.Plavan): (see section 3.10.3, section 4.1, section 4.2.1,) - date/time of events: now called more generally “timestamp” instead of posting time (the @post-time name was clearly not appropriate anymore as XTemp now handles ad-hoc event structures). event/@post-time → event/@timestamp eventbind/@postime → eventbind/@timestampexpr - removed match/@max attribute (unclear semantics, unnecessary).
WD 0.93			Approved as a CSD on March 29, 2011. Updates compared to 0.91: - various editorial changes and improvements. - split former section 3 by moving the VP-time and "concurrency" semantics in new section 3, leaving Section 4 exclusively for language syntax. - renamed "sleep" as "wait". - in addition: propose to rename "synchronized" and "non-synchronized" invocations, as "blocking" and "non-blocking" invocations, due to the overloaded meaning of "synchronized" as in CSP concurrency model (where even in "non-blocking" invocations, some event-based synchronization could take

Rev	Date	By Whom	What
			place between scriplets.) - added a definition of "concurrency" according to XTemp (i.e. uniquely depending on VP-time) - date/time of events: now called more generally timestamp instead of posting time (the @post-time name was clearly not appropriate anymore as XTemp now handles ad-hoc event structures). - aded a characterization of XTemp in terms of concurrent languages (more like CSP model than Actor model) end of 3.2.2.
WD 0.93	04/20/11	Jacques Durand	- schema updates (fixing lack of mixed content support, in some elements, missing operation “mask”, format of some date/time attribtues in <start> and <catch> did not support expressions) - related changes in specification body and RelaxNG descriptions.
WD 02	08/09/11	Jacques Durand	- few discrepancies between specification and schem fixed (cardinalities of <filter>, <match>), including typo fixed (<lval-next> → <lvar-next>) - some Appendices were not reflected in TOC. - in Section 5, and Appendix about schema, more explicit mention of optional status of XTemp event schema.
WD 02	09/23/11	Jacques Durand	- moved Reference section from appendix to Section 1, made it non-normative.