openXES

Developer Guide

Christian W. Günther
christian@fluxicon.com

Library version: 1.0 RC7
Revision: 7

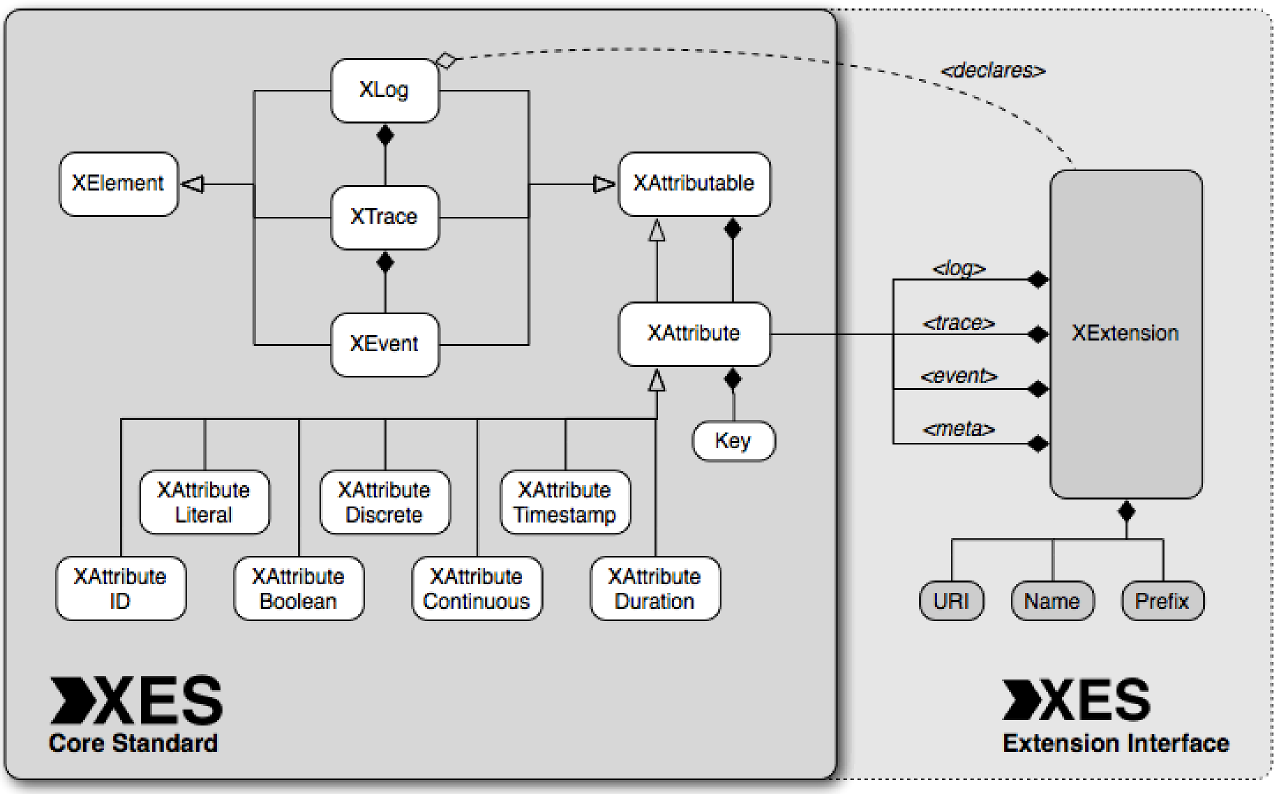
November 14, 2009

The XES Meta-model

Introduction

Event logs, as they occur in practice and research, can take a plethora of different forms and instantiations. Every system architecture that includes some sort of logging mechanism has so far developed their own, insular solution for this task. In order to be able to capture all sorts of event logs appropriately, and with as little distortion of semantics as possible, the XES meta model for event logs adopts a generic approach.

Only those elements which can be identified in virtually any setting are explicitly defined by the standard. All further information is deferred to optional attributes, which may be standardized (in terms of their semantics) by external extensions. Below you can see a UML 2.0 class diagram of the main interface components of the XES standard.



The left part of the diagram is devoted to the XES Core Standard, which needs to be implemented and fulfilled by any XES-compliant event log. Extensions to this core standard are realized through the XES Extension interface, which is explained on the right side of the diagram. In order to make it more easy to distinguish XES components from other parts of a serialization or program implementation, it has been chosen to use an ‘X’ prefix for all component names.

In the following, the components identified in the XES standard, and their relations are introduced and explained in more detail.

1. Core Standard

The XES Core Standard defines the basic structure of an event log. Its objective is to provide a generic framework onto which all event log meta-models found in practice can be mapped with relative ease, without assuming a specific field of application, or any purpose of the event logs whatsoever.

Note that the XES core standard describes only the structure of event log data. The actual information conveyed by an event log is stored in attributes, which can be attached to all layers of the core standard. Any standardization of attributes, which defines their semantics on an inter-applicational scale, is not part of the core standard, but is rather left to various extensions of the standard.

The components identified as part of the core standard, and their relations, are introduced in the following.

XElement

This interface provides a common base for all classes which make up the XES core structure.

XLog

In the XES meta-model, a log is the top-level element for an XES model. It is a container for a set of traces (see: XTrace), which have been derived from the same type of behavior. In some environments, this means that a log contains traces from executing one process definition. Also, the log is used for defining the set of extensions which are used in the respective structure (see: XExtension).

Generally, all traces within one log are assumed to be related to the same type of behavior, and describe the same (defined or imaginary) process.

XTrace

A trace is an ordered set, or list, of events. These events are supposed to have been observed during a single execution of a process, in their given order.

Note that, while consecutive events may have been observed at the same time, in general the trace is supposed to represent a total order on events. Partial orders over events within the same trace can only be imposed by extensions (See: Extension Interface), for example an extension with timestamps for events may thusly impose a partial order over these. In any case, it is illegal for a trace to feature events in reverse order, e.g., when an event having occurred later precedes an earlier event in the trace.

XEvent

An event corresponds to a single, concrete observation of any activity, in one instance of a process. Events must be part of a trace, which bundles an execution instance. They are generally assumed to be atomic, i.e., they occur at a certain point in time, but over an infinitely small amount of time.

XAttributable

As it has been already mentioned above, the XES core standard does not assume any particular information in event logs, but is generally limited to defining the structure of such event logs. All information stored in XES event logs is represented by attributes, which can be attached to any part of the meta model.

All components in the core standard structure, i.e. XLog, XTrace, and XEvent can be equipped with attributes for storing information. They implement the XAttributable interface, which defines respective access methods for attributes.

Further, even attributes themselves implement the XAttributable interface, i.e., one may attach so-called meta-attributes to other attributes. In this manner, the XES meta-model allows for the definition of tree structures with arbitrary depth, for storing information in event logs. *It is not allowed for attributes to be in circular attachment, i.e., if attribute A is (potentially transitively) attached to a higher-level attribute B, there may be no transitive attachment of B to A in any form.*

XAttribute

In general, attributes in the XES meta model are generic data fields which may be attached to any element in the core standard structure. Extensions define attributes with a fixed, generally understood semantics. These defined attributes are, however, generally handled in the same manner as generic attributes.

Every attribute is identified by a string-based key, which should be unique within the attributes attached to a single node in the XES core standard tree. Attributes in XES are strongly typed. This means that, in order to access the value of an attribute in a meaningful manner, one needs to interact with one of the strongly typed sub-interfaces of XAttribute.

The current set of attribute types is represented by the following set of XAttribute sub-interfaces:

- **XAttributeLiteral:** Attributes of this type have a literal, i.e. string-based, value.
  The keyword used for this type, e.g. in serializations of the XES format, is “LITERAL”.
- **XAttributeBoolean:** Attributes of boolean type can have a value of ‘true’ or ‘false’.
  The keyword used for this type is “BOOLEAN”.
- **XAttributeDiscrete:** Attributes of discrete type contain integer values with long precision.
  The keyword used for this type is “DISCRETE”.
- **XAttributeContinuous:** Attributes of continuous type contain floating-point values with double precision.
  The keyword used for this type is “CONTINUOUS”.
- **XAttributeTimestamp:** A timestamp attribute contains a UNIX timestamp, as a long-precision integer of milliseconds since 01/01/1970. The keyword used for this type is “TIMESTAMP”.
- **XAttributeDuration:** Attributes of duration type contain some temporal duration, given in milliseconds in long integer precision. The keyword used for this type is “DURATION”.
- **XAttributeID:** Attributes of ID type contain a unique ID, commonly implemented and encoded as UUID.
  The keyword used for this type is “ID”.

While accessing and modifying concrete attribute values is only available via the strongly typed sub-interfaces listed above, the value of every attribute can also be obtained in a string-based form.

2. Extension Interface

The XES core standard is mainly concerned with defining the general structure of event log data. The actual information contained in event logs is to be stored in attributes, which do not have any semantics, or generally understood purpose.

For defining such semantics for event log information, the XES standard uses the extension interface. This means allows to define explicit extensions to the core standard, which define a fixed set of attributes for any level on the core standard structure. Attributes which are defined by an extension are thereby standardized, and are generally understood to have a specific interpretation.

XExtension

An extension defines five distinct sets of attributes for an event log, all of which may potentially be empty. Since extensions use actual XAttribute instances as prototypes for defining their attributes, this definition includes the type of attributes. The five sets of attributes correspond to the five levels of abstraction in the XES core standard structure:

- **Log attributes:** May be attached to logs, describing information about the process being executed, the system it has been executed on, etc.
- **Trace attributes:** May be attached to traces, to describe information about one specific execution instance of the process.
- **Event attributes:** May be attached to single events, describing any kind of informational dimension related to observing a process in execution.
- **Meta attributes:** May be attached to other, higher-level attributes, in order to describe them further into detail.

Besides defining these sets of attributes, extensions are further characterized by a name in string form, which should be unique for practical reasons. This name should be relatively short, but adequately describe the purpose, or informational dimension, of the extension.

Further, each extension is identified by a unique URI. This URI should point to a definition file of the extension stored on a web server, formatted in the XESEXT standard (see below).

Finally, each extension is identified by a unique prefix string, which precedes the keys of all defined attributes. For example, the ‘role’ attribute of the organizational extension, which defines the prefix ‘org’, will be referred to as ‘org:role’.

All extensions used in an XES event log data structure must be declared in the enclosing log, which maps all used attribute prefixes to the corresponding URIs of their extensions.

Implementations for the XES standard must be able to automatically construct a stub implementation for each encountered extension, whether this extension is standardized, known beforehand, or previously unknown. XES implementations may use the XESEXT definition of an extension for constructing these stubs appropriately.

The format of XESEXT definitions is described in the following.

3. The XESEXT Format for Defining XES Extensions

Each extension to the XES core standard must provide its definition globally over the internet, by storing a definition file in the XESEXT format on a publicly available web server. These files can be used by XES implementations to transparently construct extension stubs for unknown extensions.

The XESEXT format is introduced and described by example of the standard semantic extension for XESEXT, which defines attributes for all levels of the core standard structure:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xesextension name="Semantic" prefix="semantic" uri="http://code.deckfour.org/xes/semantic.xesext">
        <log>
                <attribute key="modelReference" type="LITERAL">
                        <alias mapping="EN" name="Ontology Model Reference"/>
                        <alias mapping="DE" name="Ontologie-Modellreferenz"/>
                </attribute>
        </log>
        <trace>
                <attribute key="modelReference" type="LITERAL">
                        <alias mapping="EN" name="Ontology Model Reference"/>
                        <alias mapping="DE" name="Ontologie-Modellreferenz"/>
                </attribute>
        </trace>
        <event>
                <attribute key="modelReference" type="LITERAL">
                        <alias mapping="EN" name="Ontology Model Reference"/>
                        <alias mapping="DE" name="Ontologie-Modellreferenz"/>
                </attribute>
        </event>
        <meta>
                <attribute key="modelReference" type="LITERAL">
                        <alias mapping="EN" name="Ontology Model Reference"/>
                        <alias mapping="DE" name="Ontologie-Modellreferenz"/>
                </attribute>
        </meta>
</xesextension>
```

Every XESEXT definition file is a valid XML file. The root tag '**xesextension**' defines the extension's meta data with three attributes to this tag:

- **name:** Defines the human-readable name of the extension.
- **prefix:** Defines the attribute key prefix to be used for attributes defined by this extension.
- **uri:** Defines the unique URI for this extension. This URI must be a URL pointing to this very XESEXT file.

The **xesext** tag can have five child element tags **log**, **trace**, **event**, and **meta**. These function as enclosing containers for attribute definitions. Attributes defined within these tags can be applied to their corresponding elements in the XES core standard structure. The actual attribute definitions are defined by the **attribute** element tag. Any **attribute** tag has a **key** attribute, defining the attribute key string, and a **type** attribute for defining the attribute's type. The type is given in the form of an upper-case keyword, and corresponds to the defined XES attribute types available.
Extension can define an arbitrary number of aliases for their defined attributes. Aliases are mappings, which assign to the defined attribute any other name of that attribute in a given mapping. For that purpose, **attribute** tags can have an arbitrary number of **alias** child tags, one for each alias. Alias tags have a **mapping** attribute, defining the mapping for which an alias is given. It is advised that extension definitions supply aliases for major language mappings (identified by the upper-case shorthand also used in top-level domains, i.e., **EN** for English, **DE** for German, **NL** for Dutch, **FR** for French, etc.) The **name** attribute of alias tags defines the value for the mapping, i.e. the actual alias used for the defined attribute in the given mapping.

For a more detailed definition of the XESEST standard, the reader is referred to Appendix B, which contains the XSD stylesheet definition of this standard.

4. Standardized XES Extensions

The XES meta-model recognizes and treats all extensions as equal, independent from their source or level of proliferation. This allows users of the format to extend it at will, in order to fit any purpose or domain setting, and thus makes XES a flexible format for all applications. Due to the flexible handling of extensions, and the attributes defined by those, the XES meta-model allows using applications to interpret also previously unknown information.

However, there are recurring requirements for information stored in event logs, which demand a fixed and universally understood semantics. For this purpose, a number of extensions have been standardized, and thus equipped with a fixed semantics. When creating logs for a specific domain, or also when designing log-analyzing techniques, one should consider using these standardized extensions, since they allow for a wider level of understanding of the contents of event logs.

In the following, the currently standardized extensions to the XES formats are introduced.

Concept Extension

The Concept extension defines, for all levels of the XES type hierarchy, an attribute which stores the generally understood name of type hierarchy elements.

Extension prefix: `concept`

Extension URI: `http://code.deckfour.org/xes/concept.xesext`

Attribute Level	Key	Type	Description
log, trace, event	name	LITERAL	Stores a generally understood name for any type hierarchy element. For logs, the name attribute may store the name of the process having been executed. For traces, the name attribute usually stores the case ID. For events, the name attribute represents the name of the event, e.g. the name of the executed activity represented by the event.
event	instance	LITERAL	The instance attribute is defined for events. It represents an identifier of the activity instance whose execution has generated the event.

Lifecycle Extension

The Lifecycle extension specifies, for events, the lifecycle transition they represent in a transactional model of their generating activity. This transactional model can be arbitrary, however, the Lifecycle extension also specifies a standard transactional model for activities. Using this extension is appropriate in any setting, where events denote lifecycle transitions of higher-level activities.

Extension prefix: `lifecycle`

Extension URI: `http://code.deckfour.org/xes/lifecycle.xesext`

Attribute Level	Key	Type	Description
log	model	LITERAL	This attribute refers to the lifecycle transactional model used for all events in the log. If this attribute has a value of “**standard**”, the standard lifecycle transactional model of this extension is assumed.
event	transition	LITERAL	The transition attribute is defined for events, and specifies the lifecycle transition represented by each event. If the standard transactional model of this extension is used, the value of this attribute is one out of: **schedule** - The activity is scheduled for execution. **assign** - The activity is assigned to a resource for execution. **withdraw** - Assignment has been revoked. **reassign** - Assignment after prior revocation. **start** - Execution of the activity commences. **suspend** - Execution is being paused. **resume** - Execution is restarted. **pi_abort** - The whole execution of the process is aborted for this case. **ate_abort** - Execution of the activity is aborted. **complete** - Execution of the activity is completed. **autoskip** - The activity has been skipped by the system. **manualskip** - The activity has been skipped on purpose. **unknown** - Any lifecycle transition not captured by the above categories.

Organizational Extension

The organizational extension is useful for domains, where events can be caused by human actors, who are somewhat part of an organizational structure. This extension specifies three attributes for events, which identify the actor having caused the event, and his position in the organizational structure.

Extension prefix: **`org`**

Extension URI: [`http://code.deckfour.org/xes/org.xesext`](http://code.deckfour.org/xes/org.xesext)

Attribute Level	Key	Type	Description
event	resource	LITERAL	The name, or identifier, of the resource having triggered the event.
event	role	LITERAL	The role of the resource having triggered the event, within the organizational structure.
event	group	LITERAL	The group within the organizational structure, of which the resource having triggered the event is a member.

Time Extension

In almost all applications, the exact date and time at which events occur can be precisely recorded. Storing this information is the purpose of the time extension. Recording a timestamp for events is important, since this constitutes crucial information for many event log analysis techniques.

Extension prefix: **`time`**

Extension URI: [`http://code.deckfour.org/xes/time.xesext`](http://code.deckfour.org/xes/time.xesext)

Attribute Level	Key	Type	Description
event	timestamp	TIMESTAMP	The date and time, at which the event has occurred.

Semantic Extension

Depending on the view on a process, type hierarchy artifacts may correspond to different concepts. For example, the name of an event (as specified by the Concept extension) may refer to the activity whose execution has triggered this event. However, this activity may be situated on a low level in the process meta-model, and be a part of higher-level, aggregate activities itself.

Besides events, also other elements of the XES type hierarchy may refer to a number of concepts at the same time (e.g., a log may refer to different process definitions, on different levels of abstractions). To express the fact, that one type artifact may represent a number of concepts in a process meta-model, the semantic extension has been defined.

It is assumed that there exists an ontology for the process meta-model, where every concept can be identified by a unique URI. The semantic extension defines an attribute, which allows to store a number of model references, as URIs, in any element of the XES type hierarchy.

Extension prefix: **`semantic`**

Extension URI: `http://code.deckfour.org/xes/semantic.xesext`

Attribute Level	Key	Type	Description
log, trace, event, meta	modelReference	LITERAL	References to model concepts in an ontology. Model references are stored in a literal string, as comma-separated URIs identifying the ontology concepts.

Identity Extension

In some applications, it is crucial to equip elements of the XES type hierarchy with unique identities. For example, one may want to attach unique identities to events in a trace, if one needs to define a partial order upon these.

Extension prefix: **`identity`**

Extension URI: `http://code.deckfour.org/xes/identity.xesext`

Attribute Level	Key	Type	Description
log, trace, event, meta	id	ID	The unique identity, encoded and implemented as UUID, of the log, trace, event, or parent attribute.

Classification Extension

OpenXES provides a flexible framework for classifying events, i.e. for defining which events refer to the same logical concept, and are thus to be considered equivalent. The classification extension provides the possibility to store classifiers attached to a log itself, so that a certain set of predefined classifications can be used based only on the log itself.

For a more detailed introduction into the classification framework, please refer to its description in the later chapter of this document, where the OpenXES implementation is described.

Extension prefix: **`class`**

Extension URI: `http://code.deckfour.org/xes/classification.xesext`

Attribute Level	Key	Type	Description
log	declaration	LITERAL	This attribute can be attached to a log element, and functions as container for all event classifier declarations. For itself, this attribute has no meaning, thus the assigned value can be arbitrary and should not be considered.
meta	standard	LITERAL	This attribute provides the enclosure for the single event classifier which is the *standard classifier* for the given log. Other than its special status, the standard attribute is considered like any other classifier enclosure attribute.

The classification extension also uses generic attributes, and attributes of other extensions, in order to encode the actual classifier logic. The following XML snippet shows an example use of this extension.

```
<log xes.version="1.0" openxes.version="1.0beta5" xmlns="http://code.deckfour.org/xes"
xmlns:lifecycle="http://code.deckfour.org/xes/lifecycle.xesext"
xmlns:time="http://code.deckfour.org/xes/time.xesext"
xmlns:class="http://code.deckfour.org/xes/classification.xesext"
xmlns:semantic="http://code.deckfour.org/xes/semantic.xesext"
xmlns:concept="http://code.deckfour.org/xes/concept.xesext"
xmlns:org="http://code.deckfour.org/xes/org.xesext" id="XID.192.168.23.100.1223410422420.0">
    <attribute key="description" type="LITERAL" value="Example Log"/>
    <lifecycle:model type="LITERAL" value="standard"/>
    <class:declaration type="LITERAL" value="Event Classifier Declaration">
        <class:standard type="LITERAL" value="MXML Legacy Classifier">
            <attribute key="MXML Legacy Classifier" type="LITERAL" value="AND">
                <concept:name type="LITERAL" value="Attribute Prototype"/>
                <lifecycle:transition type="LITERAL" value="Attribute Prototype"/>
            </attribute>
        </class:standard>
        <attribute key="alternativeClassifier2" type="LITERAL" value="Resource Classifier">
            <org:resource type="LITERAL" value="Attribute Prototype"/>
        </attribute>
        <attribute key="alternativeClassifier1" type="LITERAL" value="Event Name Classifier">
            <concept:name type="LITERAL" value="Attribute Prototype"/>
        </attribute>
    </class:declaration>
```

The **class:declaration** attribute is the top-level container for all classifier declarations, attached directly to the **log** element. It contains the classifier declaration enclosure attributes, which contain the actual classifier definition. Classifier enclosures are generic attributes, or one of them may be the standard classifier's enclosure, using the **class:standard** key / tag name instead. The value of the enclosure attribute, which has literal type, contains the name of the declared classifier.

An enclosure contains the definition of the actual classifier. For simple, attribute-based classifiers, it contains a prototype of that attribute. In the example above, the "Resource Classifier", for example, features only the defining **org.resource** attribute as a prototype.

Classifiers can also be composite, i.e. be defined as a logical concatenation of attribute-based classifiers. An example for composite classifiers, above, is the **MXML Legacy Classifier** (which is also the standard classifier in this example). It is defined by a generic attribute of type literal, with the logic (in the above case: "AND") applied to the enclosed, lower-level classifiers (in this case, classifiers based on the attributes **concept:name** and **lifecycle:transition**).

5. XES Serialization

The core specification of the XES standard is focused on defining a generic meta-model for event log data, which is important for different applications that need to interoperate on the same set of data. However, for interoperability it is also of high importance that the serializations, i.e. storage formats, of a standard are strictly defined and universally understood.

In the following, the standardized serializations for the XES meta-model are defined.

XML Serialization Format for XES

The XML Serialization for XES (here-forth referred to as XES.XML) needs to be implemented by every solution to be compatible with the XES standard, since it represents the most portable and durable storage method. This format is composed in a relatively straightforward manner, reflecting the XES meta-model wherever possible.

An example XES data structure in its XML serialization is given in the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<log xes.version="1.0" openxes.version="1.0alpha1" xmlns="http://code.deckfour.org/xes"
xmlns:concept="http://code.deckfour.org/xes/concept.xesext"
xmlns:lifecycle="http://code.deckfour.org/xes/lifecycle.xesext"
xmlns:time="http://code.deckfour.org/xes/time.xesext"
xmlns:semantic="http://code.deckfour.org/xes/semantic.xesext"
xmlns:org="http://code.deckfour.org/xes/org.xesext">
        <lifecycle:model type="LITERAL" value="standard"/>
        <concept:name type="LITERAL" value="Sample Process"/>
        <attribute key="process.version" type="CONTINUOUS" value="2.4"/>
        <trace>
                <concept:name type="LITERAL" value="instance_1"/>
                <event>
                        <concept:name type="LITERAL" value="Open File"/>
                        <time:timestamp type="TIMESTAMP" value="1157467032000"/>
                        <lifecycle:transition type="LITERAL" value="complete"/>
                        <org:resource type="LITERAL" value="John"/>
                </event>
                <event>
                        <concept:name type="LITERAL" value="Update Address"/>
                        <time:timestamp type="TIMESTAMP" value="1157467053000"/>
                        <lifecycle:transition type="LITERAL" value="complete"/>
                        <org:resource type="LITERAL" value="Mary"/>
                </event>
```

```
			<event>
				<concept:name type="LITERAL" value="Close File"/>
				<time:timestamp type="TIMESTAMP" value="1157467093000"/>
				<lifecycle:transition type="LITERAL" value="complete"/>
				<org:resource type="LITERAL" value="John"/>
			</event>
		</trace>
		<trace>
			<concept:name type="LITERAL" value="instance_2"/>
			<event>
				<concept:name type="LITERAL" value="Open File"/>
				<time:timestamp type="TIMESTAMP" value="12345467032000"/>
				<lifecycle:transition type="LITERAL" value="complete"/>
				<org:resource type="LITERAL" value="John"/>
			</event>
			<event>
				<concept:name type="LITERAL" value="Close File"/>
				<time:timestamp type="TIMESTAMP" value="1157467053000"/>
				<lifecycle:transition type="LITERAL" value="complete"/>
				<org:resource type="LITERAL" value="Frank"/>
			</event>
		</trace>
</log>
```

The root element of every XES.XML file is the **log** tag, representing the single log element for every serialization. Note that every log needs to be stored in a separate serialization. The log tag has one mandatory attribute:

- **xes.version**: Represents the version of the XES standard, and thus its XML serialization, which this file uses.

Further, the log tag refers to the extensions used in this file, by mapping their URI to their respective prefix string, in the form of an XML namespace definition. It is mandatory that every extension used in a file is also referenced in the log tag, in this manner.

The structure of an XES.XML file is tightly oriented at the XES meta-model. Log tags contain a sequence of (at least one) **trace** children tags; Trace tags contain a sequence of (at least one) **event** children tags.

Every element of this structure can, further, contain a sequence of **attribute** tags, where this sequence may also be empty. The sequence of attribute tags must precede any children tags within the XES type hierarchy (i.e., the attribute children of a trace tag must precede all of its event children).

Attribute tags have three mandatory (XML tag) attributes each:

- **key**: The unique key of the described attribute.
- **type**: The type keyword of the attribute's value, as one out of **LITERAL**, **BOOLEAN**, **CONTINUOUS**, **DISCRETE**, **TIMESTAMP**, and **DURATION**.
- **value**: The value associated with the attribute, in a string-based representation.

Depending on their type, attribute values are encoded as strings according to the following specification:

- **LITERAL**: Attributes with literal type encode their values as-is, i.e., as the string value they contain.
- **BOOLEAN**: Attributes with boolean type encode their values in a string that can be either "*true*" or "*false*".
- **CONTINUOUS**: Attributes with continuous type encode their values in the format specified for the Java Double.toString() method (e.g., "*0.483907*", "*NaN*", "*Infinity*", or "*4.253E5*").
- **DISCRETE**: Attributes with discrete type encode their values as string-based integer representation (e.g., "*383747483*").
- **TIMESTAMP**: Attributes with timestamp type encode the UNIX timestamp (e.g., milliseconds since 01/01/1970) they represent, in the same form as discrete attributes.
- **DURATION**: Attributes with duration type encode their values in milliseconds, in the same form as discrete attributes.
- **ID**: Attributes with ID type encode their values as defined in the Java JDK here: http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html#toString()

Note that the above encoding schema is the same as the one used for obtaining a string-based representation of any attribute (i.e., via the Java *toString()* method call).

Attributes defined by extensions (as referenced in the log tag) are composed differently. Their respective element name is composed according to the pattern **<extension_prefix:attribute_key>**. Thus, the "resource" attribute of the organizational extension would be represented by a **<org:resource>** element. The attributes required for an extension attribute tag are equal to those of generic attributes, while omitting the key attribute (which is already implicitly included in the element name).

As specified in the XES standard's meta-model, attributes of all kinds (generic and extension) can have children attributes, which are represented as children elements of the respective attribute tag.

For a more detailed definition of the XES.XML serialization standard, the reader is referred to Appendix A, which contains the XSD stylesheet definition of this standard.

The OpenXES library

1. Introduction

The XES meta-model has been purposefully and carefully designed to be independent of any implementation. OpenXES is a reference implementation which has been designed to adhere to the following goals:

- To be fully compliant to the XES standard in every aspect.
- To be straightforward to use and easy to integrate by developers.
- To provide the highest performance for event log data management and storage.
- To serve as clear and understandable reference implementation for other implementations of the standard.

In the remainder of this section the OpenXES implementation of the XES standard is described in more detail. It is supposed to serve as a valuable, high-level guideline for developers seeking to implement systems that require event log storage, management, serialization, or analysis. Every developer using the OpenXES implementation is strongly advised to carefully study the information given in this document, and to refer to the code and its meta-documentation (i.e., Javadoc) for more detailed information.

We hope that OpenXES enables you to implement your solution in the best possible manner, provides you with powerful tools for your application domain's requirements, and makes development easy, quick, and fun to do.

2. XES Model Type Hierarchy

This section introduces the XES Model Type Hierarchy, as it is implemented in the OpenXES library.

XID

IDs are an integral part of the XES standard, since they are mandatory attributes for every element of the model type hierarchy. In OpenXES, the ID management is implemented according to the XES standard, and is represented in the package **org.deckfour.xes.id**.

- The **XID** class encapsulates the ID's ingredients transparently, and provides tools for reading them from their string representation and data streams.
- The **XIDFactory** class provides means for generating unique IDs, basing them on the system's current state. Factory methods provided by this class are the recommended way to create XID instances.

Attribute Management

Every element of the XES model type hierarchy can be equipped with attributes, even attributes themselves. Thus, attribute management takes an important role also in the OpenXES implementation.

The important interfaces for this task are situated in the package **org.deckfour.xes.model**, while several implementations may exist.

- The **XAttribute** interface defines the basic skeleton for attributes in OpenXES, including access to the attribute’s key and extension (if available). The type of an attribute, as well as access to the value of an attribute, is provided by the strongly typed sub-interfaces of XAttribute as follows.
    - The **XAttributeLiteral** interface extends the XAttribute interface with strongly typed methods to access and modify the string-based value.
    - The **XAttributeBoolean** interface extends the XAttribute interface with strongly typed methods to access and modify the boolean value.
    - The **XAttributeContinuous** interface extends the XAttribute interface with strongly typed methods to access and modify the double-precision floating point value.
    - The **XAttributeDiscrete** interface extends the XAttribute interface with strongly typed methods to access and modify the long-precision integer value.
    - The **XAttributeTimestamp** interface extends the XAttribute interface with strongly typed methods to access and modify the timestamp value. It includes both methods based on the *java.util.Date* class, as well as raw methods directly accessing the long-precision integer UNIX timestamp value (in milliseconds).
    - The **XAttributeDuration** interface extends the XAttribute interface with strongly typed methods to access and modify the duration value, interpreted as a long-precision integer in milliseconds.
    - The **XAttributeID** interface extends the XAttribute interface with strongly typed methods to access and modify the XID value.
- The **XAttributeMap** interface defines a container for attributes. It is not desirable to attach attributes directly to their respective model type hierarchy elements, both for reasons of clarity and for implementation efficiency. Every object that can take attributes stores and manages these within an instance of this interface.
- The **XAttributable** interface defines capabilities of model type hierarchy elements, which can be equipped with attributes.

Model Type Hierarchy

The actual model type hierarchy elements of the XES standard are also defined by interfaces which can be found in the package **org.deckfour.xes.model**. They are based on standard Collection interfaces, as they can be found in Sun's JDK (cf. http://java.sun.com/).

- The **XElement** interface is extended by all elements of the XES model type hierarchy in OpenXES. It defines that every element needs to have a XID, be attributable (cf. previous section), be cloneable (cf. the following section).
- The **XLog** interface extends the XElement interface. Also, it extends the generic **Set<XTrace>** interface for accessing and modifying the set of contained trace elements.
- The **XTrace** interface extends the XElement interface. Also, it extends the generic **List<XEvent>** interface for accessing and modifying the ordered list of contained log elements.
- The **XEvent** interface extends the XElement interface.

Accessing and modifying elements of an XES model type hierarchy within OpenXES is straightforward for any developer who is familiar with the Java Collection interfaces, since they all extend and implement their functionality. For creating your own XES model type hierarchies in OpenXES, or for topics of serialization and deserialization, please consult the later section dedicated to that topic.

Cloneability

Every element of the XES model type hierarchy in OpenXES extends the **Cloneable** interface in Java. In fact, most implementations in OpenXES extend this interface. This allows for a correct, straightforward, and clearly defined way of duplicating elements of an XES model within OpenXES. For more information about cloning, or the Cloneable interface, please consult the official Java documentation from Sun at http://java.sun.com/.

Building XES Models

For many elements of the XES model type hierarchy, as introduced in earlier sections, there exist multiple, alternative implementations, for various reasons. Developers should generally not worry about the concrete implementation used for their model type hierarchies, and use the factory methods provided.

Factory methods for XES model type hierarchy are provided by the class **XModelFactory**, which can be found in the package **org.deckfour.xes.model.factory** in OpenXES.

Whether you construct an XES model from scratch in OpenXES, or you want to add further elements to an existing model, the preferred way to construct new elements is to use these factory methods. Some methods allow you to provide hints on the desired implementation characteristics. It is generally safe to use the generic factory methods, since these are always guaranteed to provide the most optimal, while safe to use, implementation for generic tasks.

For more information about available implementations, please consult later sections dedicated to that topic.

3. Extensions

Extensions to the XES standard are first class citizens in the OpenXES library. It provides implementations for all standard extensions to XES, as well as it allows developers to easily add convenient implementations for their own, proprietary extensions to the standard.

Extension Management

All classes which deal with the fundamentals of extensions in OpenXES, as well as those who are concerned with managing extensions in general, can be found in the package **org.deckfour.xes.extension**.

- **XExtension** is the base class for all XES extensions within OpenXES. It defines the basic capabilities of an extension representation within the library, and is used for generic extension stubs. Extensions can be queried for their standard attributes (prefix, name, URI), as well as for the attributes they define on each level of the XES type hierarchy.
- **XExtensionManager** implements extension management in OpenXES. It is implemented as a singleton class, which organizes and manages all extensions currently known to the library framework. Proprietary extensions should be registered with the extension manager, before any actual work is being done (i.e., preferably on application startup). The extension manager also transparently manages a cache for dynamically loaded extensions, which were previously unknown to the system, and thus optimizes network access and library performance for extension resolution.
- The class **XExtensionParser** is implemented as a singleton as well. It can be used to create extension stubs from XESEXT definitions for XES extensions, from a variety of sources (files, URIs, etc.). This parser is mainly used by the extension manager, in order to resolve and satisfy availability of previously unknown extensions which are referenced in loaded serializations of XES models.

The specific implementation of extension management in OpenXES is rather involved. Thus, any developer planning to use this system other than transparently (which should work for the majority of OpenXES applications) should consult the source code and adjacent documentation, available from http://code.deckfour.org/xes/.

Standard Extensions

OpenXES provides convenient implementations for the standard extensions defined for XES. These implementations are realized as singleton classes, and can be found in the package **org.deckfour.xes.extension.std**.

Standard extension implementations provide methods for accessing and modifying attributes defined by the respective extension in a convenient, strongly-typed manner. Currently, the following set of standard extensions is implemented in OpenXES.

- **XConceptExtension**
- **XLifecycleExtension**
- **XOrganizationalExtension**
- **XSemanticExtension**
- **XTimeExtension**
- **XClassificationExtension**

OpenXES also provides a convenience wrapper for the XEvent interface, which provides easy access to the standardized extension's attributes directly. This wrapper is implemented in the class **XExtendedEvent**, which is derived from the standard XEvent interface. A static method can be used to wrap regular events in instances of this class, and thus provide more extended and typed access to this event's attributes.

The use of the XExtendedEvent is generally discouraged, since it may create the illusion that there is a standard set of attributes which is both *always* available in every XES model, and which is *never* extended by other attributes. Users of the XExtendedEvent wrapper should be aware that every attribute in XES must be considered on an equal level of importance, and should design their solutions appropriately.

4. Classification and Info

In the XES standard model, two events (as any other elements of the type hierarchy) are only equal, if they have the same ID, i .e., if they are indeed the exact same instance. For many event log analysis applications, however, one will want to have a somewhat more extended and refined methodology for defining equivalence between events.

The classification architecture within OpenXES satisfies this desire, by providing a generically configurable and extensible mechanism for defining event equivalence. Based on this mechanism, the log info in OpenXES can be used to obtain general, high-level information about event logs.

Event Classification Architecture

The event classification architecture in OpenXES provides an open, configurable mechanism to define a classification of some sort over the set of events in a log. Thereby, it projects an equivalence relation on events, generating subsets of events which are considered to be equivalent, i.e., to refer to the same semantic concept.

All interfaces and classes implementing the event classification architecture can be found in the package **org.deckfour.xes.classification**.

- The interface **XEventClassifier** is the cornerstone of the classification architecture. It defines a method for determining, whether two events belong to the same event class (i.e., whether they are equivalent). Further, it requires that the classifier be able to assign a unique class identity string to each event. Any event classifier can also be queried for the set of XAttribute instances, on which the equivalence relation of the classifier is based. This can be useful for generating events that are classified differently than a known set of events. Event classifiers can be assigned a custom name, to that they conform to the understanding of their implemented classification in a given environment.
- The class **XEventClass** represents a class, or type, of events, i.e., a set of events which are considered equivalent. It is equipped with a unique identifier string, and can be queried for size, i.e., the number of concrete events represented by that class. Also, each event class has an integer index which is unique among comparable event classes. This index can be used for addressing event classes in arrays or matrices, by applications using this feature.
- The class **XEventClasses** manages a set of event classes, i.e., XEventClass instances. A set of event classes, as represented by an instance of XEventClasses, can be created using a static factory method of this class. This class generation is based on an actual event log, whose events are being analyzed, and on an event classifier, used to determine the actual event classes from the events.

Event classifiers can take many forms, and arbitrary implementations, as long as they satisfy the XEventClassifier interface. In OpenXES, classifiers are either based on event attributes, or are composite classifiers, i.e., based on a set of lower-level classifiers.

- The **XEventAttributeClassifier** class can be configured with any event attribute prototype. It will then implement a classifier which considers two events as equivalent, if their respective value for that attribute is the same. If one of the respective events does not have the defining attribute, and the other one has, they will be considered not equivalent. If both events do not have the defining attribute, they are considered equivalent.
  In OpenXES, a number of standard event classifiers are provided. They are based on the XEventAttributeClassifier, and configured with attributes defined by the standard extensions to the XES standard.
  - **XEventLifeTransClassifier**: considers two events as equivalent, if they represent the same lifecycle transition, as defined by the Lifecycle extension.
  - **XEventNameClassifier**: considers two events as equivalent, if they have the same name, as defined by the Concept extension.
  - **XEventResourceClassifier**: considers two events as equivalent, if they have been triggered by the same resource, as defined by the Organizational extension.
- OpenXES further provides *composite event classifiers*, which impose a boolean logic on a set of lower-level classifiers. These composite classifiers allow users of the OpenXES library to design custom notions of event equivalence, based on arbitrary combinations of event attribute equality.
  - **XEventAndClassifier** can be instantiated with a list of other, lower-level classifiers, and implements the boolean AND logic. Only if all these classifiers thus consider two events as equivalent, this classifier will also consider them as equivalent.

Log Info

The log info in OpenXES provides developers with a straightforward manner to obtain high-level, aggregate information about an event log. Many of its features are based on the event classifier architecture, thus it is important to use an appropriate classifier for the log info.

All interfaces and classes implementing the log info architecture can be found in the package **org.deckfour.xes.info**.

- The **XTimeBounds** interface is a utility used by the log info. It stores a span of time, with a start and end date and time.
- The **XAttributeInfo** class provides aggregate information about the types of attributes used in a log. It is provided by the XLogInfo class (see next point) on all available levels of abstraction, i.e., log, trace, event, and meta. Developers can request the relevant XAttributeInfo instances from the log info to get information about the attributes used, attributes of a certain value type, etc.
- The **XLogInfo** interface defines a log info for OpenXES, and provides access to all aggregated information.
- The **XLogInfoFactory** class provides static factory-pattern creation methods for creating XLogInfo instances from a log. XLogInfo instances can be derived from a log by using one of the provided, static factory methods of this class. When creating a log info, one may optionally configure it with an event classifier, defining event equivalence (cf. previous section). If no explicit classifier is given, the standard classifier will be used. The standard classifier considers events as equal, if they have the same name and lifecycle transition.

The log info in OpenXES represents the summarized event log at the point of creation. If the log is modified afterwards, the information provided by the log info may no longer be accurate. Thus, log info instances should be created when they are needed, i.e., as late as possible.

The log info provides access to the following information.

- A reference to the summarized event log instance.
- Total number of events in the log.
- Total number of traces in the log.
- The set of event classes represented in the log (as defined by the optional classifier above).
- A set of event classes representing the resources found in the log (as defined by the Organizational extension).
- A set of event classes representing the event names found in the log (as defined by the Concept extension).
- A set of event classes representing the lifecycle transitions found in the log (as defined by the Lifecycle extension).
- The time boundaries of the complete log (as defined by the Time extension).
- The time boundaries of any trace in the log (as defined by the Time extension).
- Attribute information (XAttributeInfo) about the log, trace, event, and meta levels in the log.

Note that the log info has been optimized to provide the most frequently used information for log analysis and profiling, while ensuring a quick scanning procedure during log info generation. For more involved log analysis, developers should implement custom solutions, which take the associated peculiarities into account.

Alias Mapping for Attributes

Another facility provided by OpenXES is alias mapping for attributes. In order to use attributes in, e.g., a GUI-based application, one may want to use more human-readable names for attributes than their, typically rather cryptic, attribute keys. In order to solve this task, OpenXES provides a global attribute alias mapping service.

All interfaces and classes which are used for implementing alias mapping can be found in the package **org.deckfour.xes.info**.

The singleton class **XGlobalAttributeNameMap** can be used to define and use attribute name alias mapping throughout the framework. Actual attribute name mappings can provide a custom name (e.g., localized, or in custom terminology) for each XAttribute. Such actual mappings can be accessed by their **XAttributeNameMap** interface. All concrete, actual mappings are managed by the global, static, and singleton XGlobalAttributeNameMap instance.

Custom applications, which expect a certain set of attributes, can use a custom attribute name map, by requesting it from the global service. This map can be used to define the actual mapping, or that mapping can be directly defined using the global attribute name map (which will delegate these requests accordingly).

It is especially important for custom extension implementations, that they define attribute name mappings for their defined attributes in their constructor (or, as early as possible). Extensions should provide custom name mappings for their defined at least for the English language (i.e., “EN” attribute name map).

5. Reading and Serializing XES Models

In this section, we introduce the capabilities within the OpenXES library for serialization, i.e., for writing an XES model to, or reading it from, any sort of data stream.

Reading XES Models From Serializations

All classes concerned with reading XES Models in OpenXES are located in the package **org.deckfour.xes.in**.

- **XesXmlParser** provides the capabilities to read XES models from their standardized XML representation, as introduced previously in this document. XES models can be read from files and regular input streams. When reading from files, the parser checks for a ".gz" extension of the file. If present, GZIP decompression is transparently added to the parsing step.
- **XesMxmlParser** provides the capabilities to read XES models from the legacy MXML format, as traditionally used in the process mining community. XES models can be read from files and regular input streams. When reading from files, the parser checks for a ".gz" extension of the file. If present, GZIP decompression is transparently added to the parsing step.

Reading XES models is generally a straightforward procedure. Any errors that may occur are reported in the form of appropriate exceptions, which are thrown by the parsing methods.

OpenXES further provides convenience facilities to monitor the parsing of any serialization. These are described in detail in a dedicated, later section of this document.

Serialization of XES Models

All interfaces and classes concerned with the serialization of XES models in OpenXES can be found in the package **org.deckfour.xes.out**.

- The **XesSerializer** interface defines the basic capabilities of any XES serialization in OpenXES. Any serialization needs to have a (human-readable) name and description, provide the filename extensions used for the respective serialization, and be able to serialize an XLog object (containing a model type hierarchy) to a standard Java output stream.
- The **XesXmlSerializer** implements the serialization of XES models to the XML serialization, standardized earlier in this document.
- The **XMxmlSerializer** implements the serialization of XES models to the legacy MXML standard. It is provided for reasons of backwards compatibility, and its use is generally discouraged.

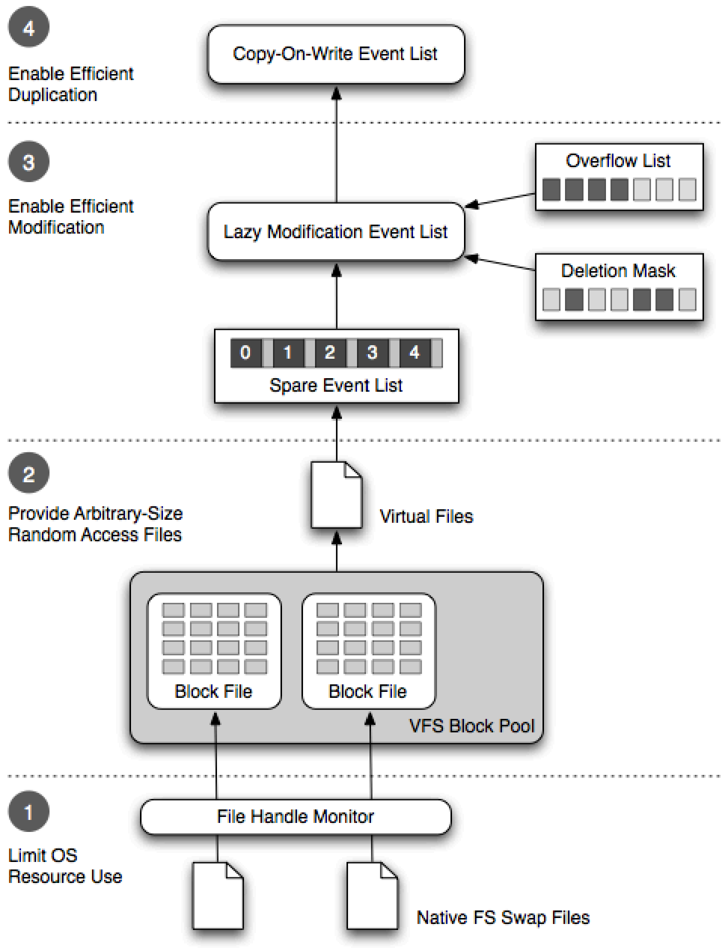
Serializing XES models is generally a straightforward procedure. Any errors that may occur are reported in the form of appropriate exceptions, which are thrown by the serializing method. The provision of meta-information, such as name, description, and file name extensions, allow using applications to easily integrate the capabilities of serializers.

6. Data Storage and Implementation Issues

This section contains information which is concerned with implementation issues of the OpenXES library. In order to provide the best possible performance, and thus enable realistic applications of OpenXES, this implementation uses a sophisticated architecture for storing and managing event log data. This storage method incurs some trade-offs, which will need to be considered when implementing solutions based on OpenXES.

The NikeFS2 Virtual Storage Layer

The actual storage of event log data in OpenXES can be optionally offloaded to disk, which frees the main memory of the using application for other tasks (e.g., analysis). In order to provide this offloaded storage in an efficient manner, OpenXES uses the NikeFS2 virtual storage layer for optimized binary data storage.



The following diagram illustrates the architecture of the disk-buffered event log data storage layer in OpenXES.

The NikeFS2 storage subsystem is based on four distinct layers, which each provide performance optimizations in a transparent manner.

- **Layer 1**: The NikeFS2 event log data storage subsystem offloads actual data to swap files, organized by the host operating system. This is the basic precondition for freeing system memory for other tasks.
- **Layer 2**: Buffer files are organized in fixed-size blocks of binary data, which are managed in a block pool. NikeFS2 provides virtual, binary file abstractions, which are backed by a set of blocks from this pool. This enables the log storage layer to provide binary files of arbitrary length in an efficient manner, much like the actual file system in an operating system.
- **Layer 3**: In this layer, the NikeFS2 system provides two optimizations in order to enable the efficient modification of disk-buffered lists of events, i.e. traces.
  - The **spare event list** is based on virtual files as provided by Layer 2. It saves a pre-defined number of bytes after each stored event in the binary stream. This allows to replace events later on in mid-stream, given that their size is no larger than that of the original event plus the spare space that has been saved. In this manner, spare event lists prevent frequent re-writing of the complete event stream.
  - The **lazy modification event list** abstractions are based on spare event lists. These lists are equipped with meta data structures, capturing modifications in the form of an overflow list, storing added events, and a deletion mask, remembering which events have been deleted from the actual stream. This enables quick modification of event lists, and saves the overhead of frequent modification of the actual binary storage stream. Once the meta data structures are saturated, the lazy modification event list is flushed, i.e. reverted to a canonical form by creating a new byte stream for backing it.
- **Layer 4**: In this layer, NikeFS2 provides copy-on-write event lists. These wrap a lazy modification event list transparently, and only point to them virtually. If a copy-on-write list is copied, the actual data is still sourced from the original event list. Only when modifications occur does this list create an actual copy. By using this technique from file system design, it significantly speeds up the creation of cheap copies, as long as modifications are infrequent.

The use of NikeFS2 storage should generally be transparent, both for the user and the developer of applications based on OpenXES. It provides significant performance improvements, and enables the analysis of large event log data sets in the first place.

However, there are some implications for development, which are discussed in the following section.

Buffered Trace and Attribute Map Implementations

OpenXES provides buffered implementations for both the XTrace and the XAttributeMap interface. These benefit from performance increases, and savings in main memory usage, and are generally encouraged for usage. However, there are a number of implications which developers of solutions based on OpenXES need to consider.

- XEvent objects, when requested from a buffered XTrace implementation, are transient objects. This means, if one requests the same event (i.e., at the same location in the same trace) two times, one will essentially receive two different objects. This is mitigated by the use of XIDs for equality tests (i.e., the equals() method).
- Related to the previous point: If an XEvent instance, which one has obtained from a buffered XTrace instance, is modified, this modification is not persistent. One will need to replace the original event by the modified one in the trace, in order to make changes persistent.
- The above points are also true for XAttribute objects requested from buffered XAttributeMap instances. Changes to these are not persistent. One will need to re-set the attribute in the map, in order to trigger its persistent storage.

These restrictions of the buffered implementation are inherent to the storage architecture used, and represent a certain trade-off for development. However, when keeping them in mind during development, a correct implementation that adheres to the specific requirements which this necessitates is usually quite straightforward to realize.

7. Utilities

The OpenXES library is mainly concerned with managing event log data. On top of that, it also includes capabilities which may be useful for developing solutions based on OpenXES, but which are not related to its core functionality. These utility capabilities are introduced in the following sections.

XStream Serialization

OpenXES includes an additional library for XStream serialization of OpenXES models. This library, *OpenXES-XStream*, includes a set of converter implementations, which enables developers to use it with XStream serialization. XStream is an efficient serialization mechanism for Java object hierarchies, which can be found at http://xstream.codehaus.org/.

The classes implementing XStream serialization for OpenXES model type hierarchy elements can be found in the package **org.deckfour.xes.xstream**. In general, for every element of the XES model type hierarchy, as implemented in OpenXES, a specific XStream converter has been implemented.

Before using XStream serialization for XES models implemented by OpenXES, developers should register the OpenXES converters with XStream. The class **XesXStreamPersistency** features a static method, which should be called before XStream serialization, in order to perform all necessary registrations of converters.

Progress Monitoring For Reading Serializations

When reading XES models from serializations in GUI based applications, one frequently wants to communicate the progress of activity to the user. For this purpose, OpenXES provides progress monitoring facilities which can be easily integrated into any application. For detailed information, the interested developer is referred to the code, which can be found in the package **org.deckfour.xes.util.progress**.

Timer

OpenXES further provides a simple timing facility, which may be of general interest for developers. It is implemented in the class **XTimer** in package **org.deckfour.xes.util**. A timer is started on creation, and can be stopped and re-started by calling appropriate methods. Further, it provides methods for querying the time passed since starting / creation, and to format this time in a human-readable, nicely formatted string.

Appendix A

XES XML Serialization Schema Definition (XSD)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
      targetNamespace="http://code.deckfour.org/xes" xmlns:xes="http://code.deckfour.org/xes">
  <!-- This file describes the XML serialization of the XES format for event log data. -->
  <!-- For more information about XES, visit http://code.deckfour.org/xes/ -->
  <!-- (c) 2008 by Christian W. Guenther (christian@deckfour.org) -->

  <!-- Every XES XML Serialization needs to contain exactly one log element -->
  <xs:element name="log" type="xes:LogType"/>

  <!-- Definition of the XES XID serialization -->
  <xs:simpleType name="XID">
    <xs:restriction base="xs:string">
      <!--
        The XID is composed of the following elements in their given
        order, separated by dots:
       * 'XID' prefix.
       * Four elements denoting the host IP of the generating system.
       * One element describing the timestamp, in milliseconds since
         01/01/1970, when the ID was created.
       * One element with a sequence number unique within that timestamp.
      -->
      <xs:pattern value="XID\.\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,}\.\d{1,}"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- Attribute types can be one of these -->
  <xs:simpleType name="AttributeValueType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="LITERAL"/>
      <xs:enumeration value="BOOLEAN"/>
      <xs:enumeration value="DISCRETE"/>
      <xs:enumeration value="CONTINUOUS"/>
      <xs:enumeration value="TIMESTAMP"/>
      <xs:enumeration value="DURATION"/>
    </xs:restriction>
  </xs:simpleType>
  <!-- Attributes have a unique key, a type, and a string-based value -->
  <xs:complexType name="AttributeType">
    <xs:sequence>
      <xs:element name="attribute" minOccurs="0" maxOccurs="unbounded" type="xes:AttributeType"/>
    </xs:sequence>
    <xs:attribute name="key" use="required" type="xs:Name"/>
    <xs:attribute name="type" use="required" type="xes:AttributeValueType"/>
    <xs:attribute name="value" use="required" type="xs:string"/>
  </xs:complexType>
  <!-- Logs may contain attributes and traces -->
  <xs:complexType name="LogType">
    <xs:sequence>
      <xs:element name="attribute" minOccurs="0" maxOccurs="unbounded" type="xes:AttributeType"/>
      <xs:element name="trace" maxOccurs="unbounded" type="xes:TraceType"/>
    </xs:sequence>
    <xs:attribute name="id" type="xes:XID" use="required"/>
  </xs:complexType>

  <!-- Traces may contain attributes and events -->
  <xs:complexType name="TraceType">
    <xs:sequence>
      <xs:element name="attribute" minOccurs="0" maxOccurs="unbounded" type="xes:AttributeType"/>
      <xs:element name="event" maxOccurs="unbounded" type="xes:EventType"/>
    </xs:sequence>
    <xs:attribute name="id" type="xes:XID" use="required"/>
  </xs:complexType>

  <!-- Events may contain attributes -->
  <xs:complexType name="EventType">
    <xs:sequence>
      <xs:element name="attribute" minOccurs="0" maxOccurs="unbounded" type="xes:AttributeType"/>
    </xs:sequence>
    <xs:attribute name="id" type="xes:XID" use="required"/>
  </xs:complexType>

</xs:schema>
```

Appendix B

XESEXT Extension Format Schema Definition (XSD)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <!-- This file describes the  serialization for extensions of the XES format -->
  <!-- For more information about XES, visit http://code.deckfour.org/xes/ -->

  <!-- (c) 2008 by Christian W. Guenther (christian@deckfour.org) -->

  <!-- Any extension definition has an xesextension root element. -->
  <!-- Child elements are containers, which define attributes for -->
  <!-- the log, trace, event, and meta level of the XES -->
  <!-- type hierarchy. -->
  <!-- All of these containers are optional. -->
  <!-- The root element further has attributes, defining: -->
  <!--   * The name of the extension. -->
  <!--   * A unique prefix string for  attributes defined by this -->
  <!--     extension. -->
  <!--   * A unique URI of this extension, holding the XESEXT -->
  <!--     definition file. -->
  <xs:element name="xesextension">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" ref="log"/>
        <xs:element minOccurs="0" ref="trace"/>
        <xs:element minOccurs="0" ref="event"/>
        <xs:element minOccurs="0" ref="meta"/>
      </xs:sequence>
      <xs:attribute name="name" use="required" type="xs:NCName"/>
      <xs:attribute name="prefix" use="required" type="xs:NCName"/>
      <xs:attribute name="uri" use="required" type="xs:anyURI"/>
    </xs:complexType>
  </xs:element>
```

```
<!-- Container tag for the definition of log attributes. -->
<xs:element name="log">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="attribute"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!-- Container tag for the definition of trace attributes. -->
<xs:element name="trace">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="attribute"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!-- Container tag for the definition of event attributes. -->
<xs:element name="event">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="attribute"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!-- Container tag for the definition of meta attributes. -->
<xs:element name="meta">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="attribute"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!-- Attribute definition, defining the key and type of the attribute. -->
<xs:element name="attribute">
  <xs:complexType>
    <xs:sequence>
     <xs:element ref="alias" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="key" use="required" type="xs:NCName"/>
    <xs:attribute name="type" use="required" type="AttributeValueType"/>
  </xs:complexType>
</xs:element>

<!--  Alias definition, defining a mapping alias for an attribute -->
<xs:element name="alias">
     <xs:complexType>
            <xs:attribute name="mapping" use="required" type="xs:NCName"/>
            <xs:attribute name="name" use="required" type="xs:string"/>
     </xs:complexType>
</xs:element>

<!-- Attribute values may have one of these types. -->
<xs:simpleType name="AttributeValueType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="LITERAL"/>
```

```
      <xs:enumeration value="BOOLEAN"/>
      <xs:enumeration value="DISCRETE"/>
      <xs:enumeration value="CONTINUOUS"/>
      <xs:enumeration value="TIMESTAMP"/>
      <xs:enumeration value="DURATION"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```