Technische Universiteit
Eindhoven
University of Technology

Den Dolech 2, 5612 AZ Eindhoven
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
www.tue.nl

Author
Christian W. Günther and Eric Verbeek

Date
October 25, 2012

Version
1.8

OpenXES

Developer Guide

1 The XES Meta-model

1.1 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.

The remainder of this Chapter introduces the XES meta model in a staged manner. First, Section 1.2 introduces the basic structure of an event log. Second, Section 1.3 adds data to this basic structure by adding attributes. Third, Section 1.4 explains which types of attributes are supported. Fourth, Section 1.5 adds event classifiers, which can be used to classify events into buckets. Fifth, Section 1.6 adds the notion of global attributes, that is, attributes that are declared to be omnipresent throughout the log. Sixth, Section 1.7 adds the extension framework, which is used to impose semantics on attributes. Last, Section 1.8 shows an overview of the standardized XES extensions.

1.2 Base structure

1.2.1 Meta-model

The base structure of the XES meta-model (see Figure 1.1) states that a log contains a non-empty collection of traces, and that a trace contains a non-empty list of events. Note that the traces in the log are considered to be unordered, whereas the events in the trace are considered to be ordered. Als note that an empty log and/or empty traces are excluded.

1.2.2 XSD schema

The XSD schema reflects the meta-model using the “LogType”, “TraceType”, and “EventType”. A log is of type “LogType”, which contains at least one trace of type “TraceType”, which contains at least one event of type “EventType”. A log also contains three mandatory attributes: “xes.version”, “xes.features”, and “openxes.version”. The “xes.version” attribute states the version of XES used in this log, the “xes.features” attribute state the XES features used by the log (currently, only the feature “nested-attributes” is supported when meta-attributes are present), and the “openxes.version” attribute states the version of OpenXES required to read the log successfully.

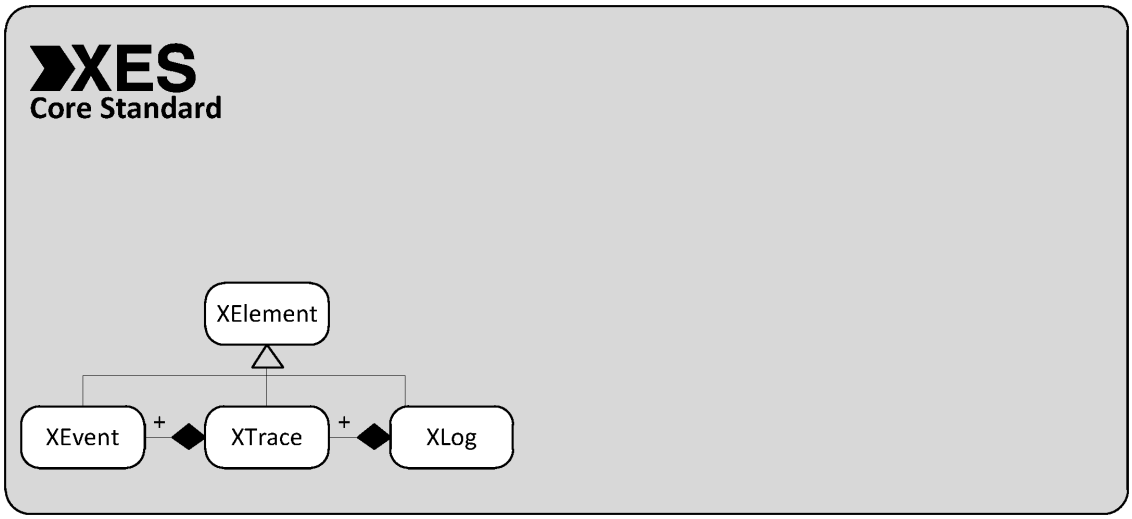

Figure 1.1: The UML 2.0 class diagram for the base structure

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

  <xs:element name="log" type="xes:LogType"/>

  ...

  <!-- Elements may contain attributes -->
  <xs:complexType name="ElementType">
    ...
  </xs:complexType>

  <!-- Logs are elements that may contain traces -->
  <xs:complexType name="LogType">
    <xs:complexContent>
      <xs:extension base="xes:ElementType">
            <xs:sequence>
              ...
              <xs:element name="trace" maxOccurs="unbounded" type="xes:TraceType"/>
            </xs:sequence>
            <xs:attribute name="xes.version" type="xs:decimal" use="required"/>
            <xs:attribute name="xes.features" type="xs:token" use="required"/>
            <xs:attribute name="openxes.version" type="xs:string" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- Traces are elements that may contain events -->
  <xs:complexType name="TraceType">
    <xs:complexContent>
      <xs:extension base="xes:ElementType">
            <xs:sequence>
              <xs:element name="event" maxOccurs="unbounded" type="xes:EventType"/>
            </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- Events are elements -->
  <xs:complexType name="EventType">
```

```
    <xs:complexContent>
      <xs:extension base="xes:ElementType">
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

</xs:schema>
```

1.2.3 Example

The example shows an example log, which contains many traces, of which the first trace contains six events. Note that no data is yet associated with the log, the traces, and the event.

```
<log xes.version="1.3" xes.features="" openxes.version="1.8" xmlns="http://www.xes-
    standard.org/">
  <trace>
    <event/>
    <event/>
    <event/>
    <event/>
    <event/>
    <event/>
  </trace>
  <trace>
    ...
  </trace>
  ...
</log>
```

1.3 Adding attributes

1.3.1 Meta-model

Attributes are added to the base structure (see Figure 1.2), and every element known so far (logs, traces, events, and attributes) are attributable, which means that they all can have attributes. Note that attributes themselves can have attributes, but that this should be declared as a feature of the log (“xes.features” attribute should include “nested-attributes”).

1.3.2 XSD schema

The XSD schema is extended with the attributes and attributables.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema ...>

  ...

  <!-- Attribute -->
  <xs:complexType name="AttributeType">
    <xs:complexContent>
      <xs:extension base="xes:AttributableType">
            <xs:attribute name="key" use="required" type="xs:token"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  ...
```

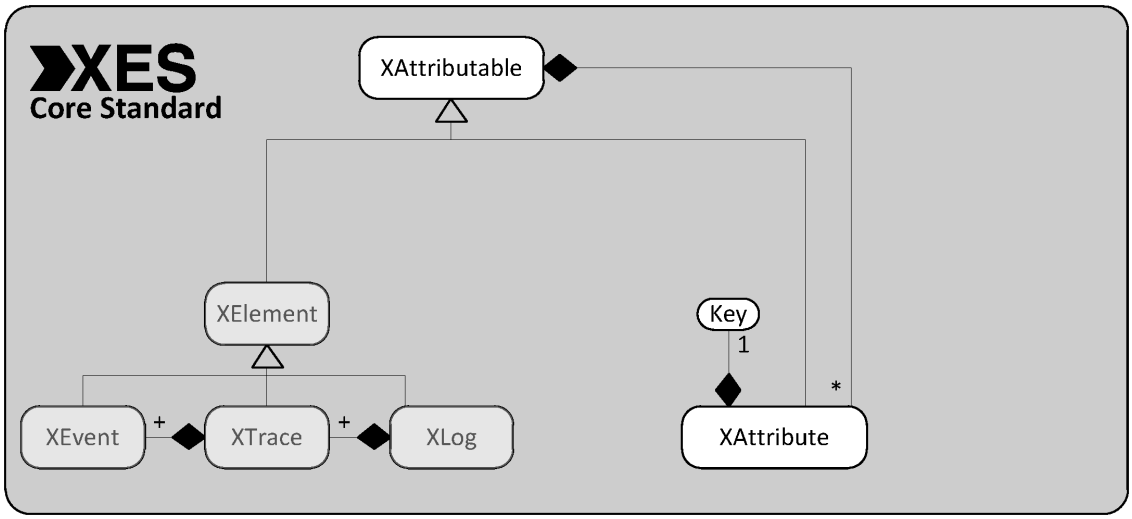

Figure 1.2: The UML 2.0 class diagram after having added the attributes

```
  <!-- Elements may contain attributes -->
  <xs:complexType name="ElementType">
    <xs:complexContent>
      <xs:extension base="xes:AttributableType"/>
    </xs:complexContent>
  </xs:complexType>

  ...

</xs:schema>
```

1.4 Attribute types

1.4.1 Meta-model

Attributes can of six different types (see Figure 1.3):

Literal: Attributes of literal type contain string values. The keyword used for this type, e.g. in serializations of the XESformat, is “string”.

Boolean: Attributes of boolean type contain boolean values (“true” or “false”). The keyword used for this type is “boolean”.

Discrete: Attributes of discrete type contain integer values with long precision. The keyword used for this type is “int”.

Continuous: Attributes of continuous type contain floating-point values with double precision. The keyword used for this type is “float”.

Timestamp: Attributes of timestamp type contain a timestamp formatted as “$yyyy$-MM-ddTHH:mm:ss.$SSSZ$”, where

$yyyy$ corresponds to the year (“1900”, “1901”, . . . , “2011”, . . .),
MM corresponds to the month of the year (“01”, “02”, . . . , “12”),

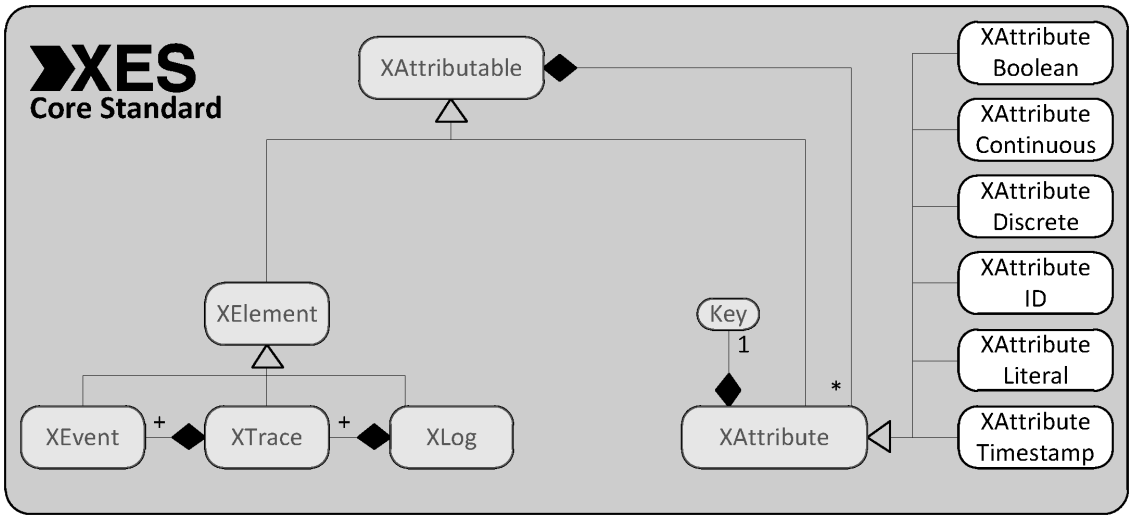

Figure 1.3: The UML 2.0 class diagram after having added the attribute types

dd corresponds to the day in the month (“01”, “02”, . . . , “31”),
HH corresponds to the hour in the day (“00”, “01”, . . . , “23”),
mm corresponds to the minute in the hour (“00”, “01”, . . . , “59”),
ss corresponds to the second in the minute (“00”, “01”, . . . , “59”),
SSS corresponds to the millisecond in the second (“000”, “001”, . . . , “999”), and
Z corresponds to the time zone relative to GMT (like “Z”, “+02:00” or “-01:30”).

The keyword used for this type is “date”.

ID: Attributes of ID type contain a unique identifier, commonly implemented and encoded as UUID. The keyword used for this type is “id”.

1.4.2 XSD schema

The XSD schema is extended with the six different attribute types. Note that the type of the value of an attribute depends on the type of the attribute. Any attributable may contain any numbers of attributes of any of the different types.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema ...>

  ...

  <xs:complexType name="AttributableType">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="string" minOccurs="0" maxOccurs="unbounded" type="
          xes:AttributeStringType"/>
      <xs:element name="date" minOccurs="0" maxOccurs="unbounded" type="
          xes:AttributeDateType"/>
      <xs:element name="int" minOccurs="0" maxOccurs="unbounded" type="
          xes:AttributeIntType"/>
      <xs:element name="float" minOccurs="0" maxOccurs="unbounded" type="
          xes:AttributeFloatType"/>
      <xs:element name="boolean" minOccurs="0" maxOccurs="unbounded" type="
          xes:AttributeBooleanType"/>
```

```
      <xs:element name="id" minOccurs="0" maxOccurs="unbounded" type="
          xes:AttributeIDType"/>
    </xs:choice>
  </xs:complexType>

  ...

  <!-- String attribute -->
  <xs:complexType name="AttributeStringType">
    <xs:complexContent>
      <xs:extension base="xes:AttributeType">
            <xs:attribute name="value" use="required" type="xs:string"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- Date attribute -->
  <xs:complexType name="AttributeDateType">
    <xs:complexContent>
      <xs:extension base="xes:AttributeType">
            <xs:attribute name="value" use="required" type="xs:dateTime"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- Integer attribute -->
  <xs:complexType name="AttributeIntType">
    <xs:complexContent>
      <xs:extension base="xes:AttributeType">
            <xs:attribute name="value" use="required" type="xs:long"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- Floating-point attribute -->
  <xs:complexType name="AttributeFloatType">
    <xs:complexContent>
      <xs:extension base="xes:AttributeType">
            <xs:attribute name="value" use="required" type="xs:double"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- Boolean attribute -->
  <xs:complexType name="AttributeBooleanType">
    <xs:complexContent>
      <xs:extension base="xes:AttributeType">
            <xs:attribute name="value" use="required" type="xs:boolean"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- ID attribute -->
  <xs:complexType name="AttributeIDType">
    <xs:complexContent>
      <xs:extension base="xes:AttributeType">
            <xs:attribute name="value" use="required" type="xs:string"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  ...

</xs:schema>
```

1.4.3 Example

The example log can now be enriched with data in the form of attributes. Apparently, the example log has been created by a tool called “Fluxicon Nitro”, the first trace is named “Case1280”, and the first event of the first trace was performed by the resource “FL” on March 15, 2010 at 7:59 AM (in timezone GMT+2).

```
<log xes.version="1.3" xes.features="nested-attributes" openxes.version="1.8" xmlns="
    http://www.xes-standard.org/">
  <string key="Creator" value="Fluxicon Nitro"/>
  <trace>
    <string key="Name" value="Case1280"/>
    <string key="Creator" value="Fluxicon Nitro"/>
    <event>
      <string key="Resource" value="FL"/>
      <date key="Timestamp" value="2010-03-15T07:59:00.000+02:00"/>
      <string key="Operation" value="Handle Email"/>
      <string key="Agent Position" value="FL"/>
      <string key="Customer ID" value="Customer 1074"/>
      <string key="Product" value="iPhone"/>
      <string key="Service Type" value="Product Assistance"/>
      <string key="Agent" value="Susi"/>
    </event>
    <event>
      ...
    </event>
    ...
  </trace>
  <trace>
    ...
  </trace>
  ...
</log>
```

1.5 Adding classifiers

1.5.1 Meta-model

Now we’ve added data to the events, but we still do not know which attributes to use when comparing events. Typically, we only want to consider a subset of attributes as a key to the event. For this sake, classifiers are added to the meta-model (see Figure 1.4). A classifier simply contains a list of attribute keys, and based on such an attribute a summary can be made of the log. Such a summary then includes how many different combinations of selected attributes exist in the log, in which frequencies, etc. In case an attribute used in some event classifier is missing, the empty value will be used in the classifier.

1.5.2 XSD schema

The XSD schema is extended with a type for the classifiers, and a log can now have any number of these classifiers. Note that the collection of attributes is defined as a combination of their keys in a single string (“keys” attribute). As the keys themselves may contain spaces, the question for the XES parser now becomes which spaces in the “keys” attribute separate two attribute keys and which spaces occur in a single attribute key. By default, the parser assumes that all spaces are separating spaces. Only if a separated key does not match any global event attribute key, then the following space will be treated as one that occurs in a single attribute key.

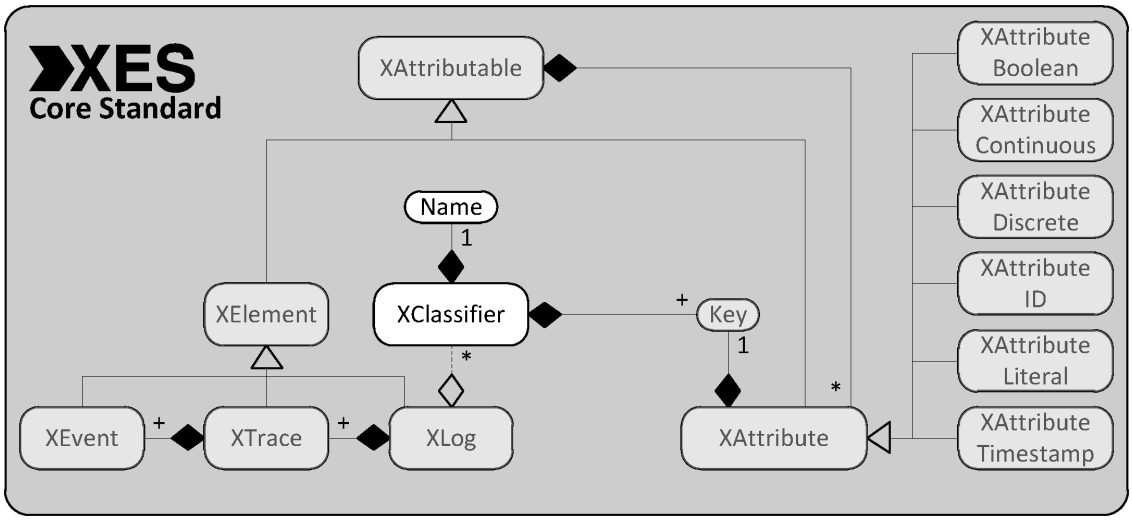

Figure 1.4: The UML 2.0 class diagram after having added the classifiers

```
...

<!-- Classifier definition -->
<xs:complexType name="ClassifierType">
  <xs:attribute name="name" type="xs:NCName" use="required"/>
  <xs:attribute name="keys" type="xs:token" use="required"/>
</xs:complexType>

<!-- Logs may contain attributes and traces -->
<xs:complexType name="LogType">
  <xs:complexContent>
    <xs:extension base="xes:ElementType">
          <xs:sequence>
            ...
            <xs:element name="classifier" minOccurs="0" maxOccurs="unbounded" type="
                xes:ClassifierType"/>
            <xs:element name="trace" maxOccurs="unbounded" type="xes:TraceType"/>
          </xs:sequence>
          ...
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

...
```

1.5.3 Example

The example shows three classifiers. The first classifier (named “Operation”) classifies the events based on only their “Operation” attribute, the second classifier (“Service Type”) classifies them on only their “Service Type” attribute (here we assume that there is a global event attribute with key “Service Type” but no global event attribute with key “Service” or “Type”), whereas the third classifier (“activity classifier”) uses both these attributes to classify them. For the first event of the first trace, the actual classifier values are “Handle Email”, “Product Assistance”, and “Handle Email+Product Assistance”. Note that “+” is used to combine the values of multiple attributes in a single classifier.

```
<log xes.version="1.3" xes.features="nested-attributes" openxes.version="1.6" xmlns="
    http://www.xes-standard.org/">
  <classifier name="Operation" keys="Operation"/>
  <classifier name="Service Type" keys="Service Type"/>
  <classifier name="activity classifier" keys="Operation Service Type"/>
  <string key="Creator" value="Fluxicon Nitro"/>
  <trace>
    <string key="Name" value="Case1280"/>
    <string key="Creator" value="Fluxicon Nitro"/>
    <event>
      ...
      <string key="Operation" value="Handle Email"/>
      ...
      <string key="Product" value="iPhone"/>
      <string key="Service Type" value="Product Assistance"/>
      ...
    </event>
    <event>
      ...
    </event>
    ...
  </trace>
  <trace>
    ...
  </trace>
  ...
</log>
```

1.6 Adding globals

1.6.1 Meta-model

Apart from being able to classify events on their attributes, it is also relevant to know which event attributes (and which trace attributes) are omnipresent in the log. Instead of having to check this by traversing the entire log, we add possible global declarations for these attributes (see Figure 1.5). These declarations come with a default value for these attributes, which is *only to be used* when having to create this attribute for a *newly created* event (trace), that is, this value *should not be used* as the value to be used when no attribute is provided for an *existing* event (trace).

1.6.2 XSD schema

The XSD schema is extended with globals that have either an “event” or “trace” scope. Logs can have up to two global declarations: one for the global event attributes and one for the global trace attributes. Please note that it is technically possible to have nested attributes in these global declarations, but that there is no use. In the global declarations section, meta-attributes will be ignored.

```
  ...

  <!-- Globals definition -->
  <xs:complexType name="GlobalsType">
    <xs:complexContent>
      <xs:extension base="xes:AttributableType">
        <xs:attribute name="scope" type="xs:NCName" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

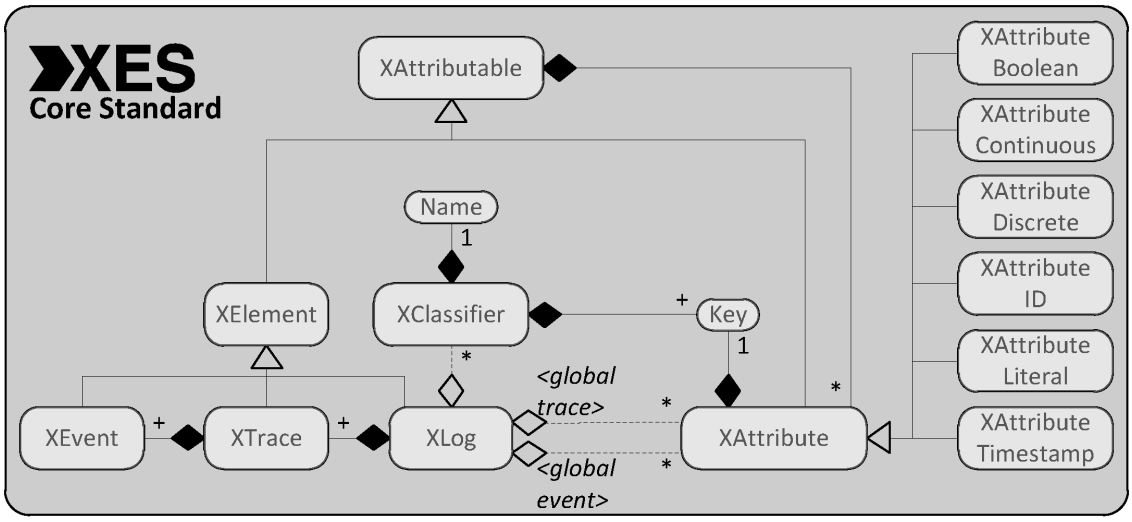

Figure 1.5: The UML 2.0 class diagram for the base structure after having added the globals

```
...

<!-- Logs may contain attributes and traces -->
<xs:complexType name="LogType">
  <xs:complexContent>
    <xs:extension base="xes:ElementType">
          <xs:sequence>
            ...
            <xs:element name="global" minOccurs="0" maxOccurs="2" type="
                xes:GlobalsType"/>
            <xs:element name="classifier" minOccurs="0" maxOccurs="unbounded" type="
                xes:ClassifierType"/>
            <xs:element name="trace" maxOccurs="unbounded" type="xes:TraceType"/>
          </xs:sequence>
          ...
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

...
```

1.6.3 Example

In the example logs, the event attribute “Name” (among others) is declared to be omnipresent, and so is the trace attribute “Name”. As a result, algorithm may assume that every event and every trace has a “Name” attribute. It is good practice to use only global event attributes used in event classifiers.

```
<log xes.version="1.3" xes.features="nested-attributes" openxes.version="1.6" xmlns="
    http://www.xes-standard.org/">
  <global scope="trace">
    <string key="Name" value="name"/>
  </global>
  <global scope="event">
    <string key="Operation" value="string"/>
        ...
    <string key="Service Type" value="string"/>
  </global>
```

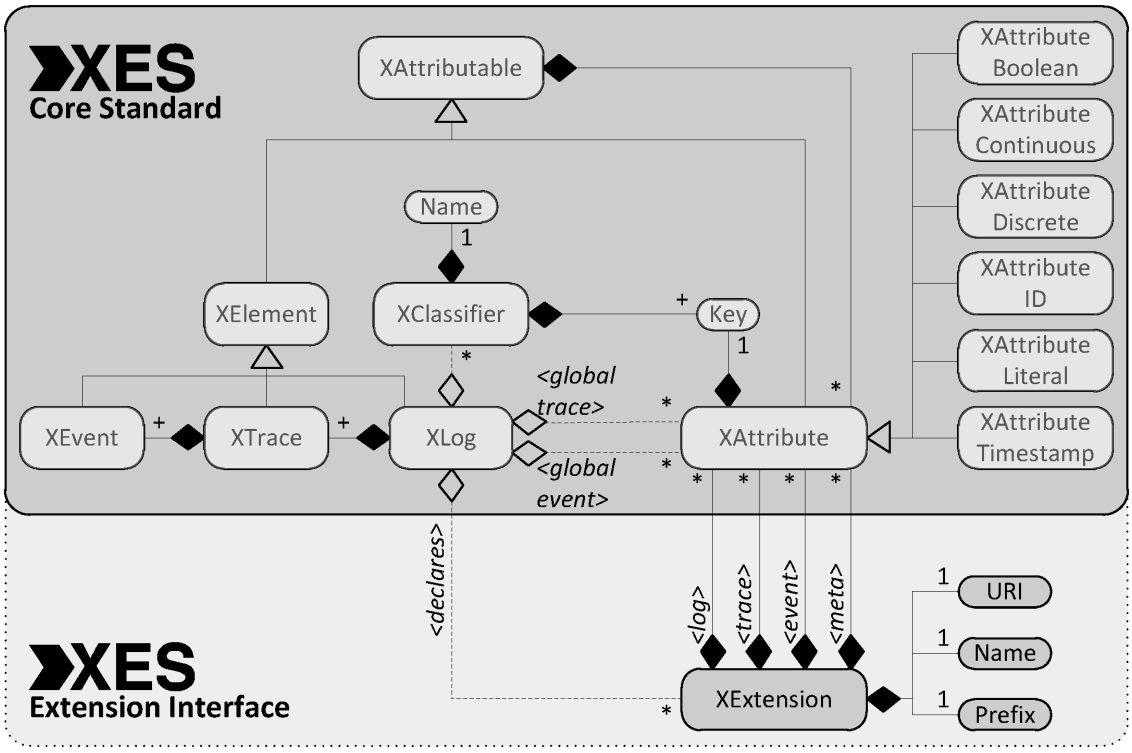

Figure 1.6: The UML 2.0 class diagram for the base structure after having added the extensions

```
  <classifier name="Operation" keys="Operation"/>
  <classifier name="Service Type" keys="Service Type"/>
  <classifier name="activity classifier" keys="Operation Service Type"/>
  <string key="Creator" value="Fluxicon Nitro"/>
  <trace>
    ...
  </trace>
  ...
</log>
```

1.7 Adding extensions

1.7.1 Meta-model

Finally, we add the extensions to the meta-model (see Figure 1.6), which provide semantics to the attributes involved. By declaring an extension in the log, and by using the attribute keys as dictated by this extension, the log provider states that the semantics of the specific attribute value corresponds to the semantics as dictated by the extension. Note that so far we have used attributes as plain key-value pairs. By linking the keys to an extension, we link the values to specific semantics.

1.7.2 XSD schema

The XSD schema is extended with the declaration of an extension, which basically contains a name for the extension, a prefix to use for the extension, and the URI where to find the details

on this extension (which will be explained in the next section). A log can declare any number of extensions.

```
...

<!-- Extension definition -->
<xs:complexType name="ExtensionType">
  <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>

...

<!-- Logs may contain attributes and traces -->
<xs:complexType name="LogType">
  <xs:complexContent>
    <xs:extension base="xes:AttributableType">
          <xs:sequence>
            <xs:element name="extension" minOccurs="0" maxOccurs="unbounded" type="
                xes:ExtensionType"/>
            <xs:element name="global" minOccurs="0" maxOccurs="2" type="
                xes:GlobalsType"/>
            <xs:element name="classifier" minOccurs="0" maxOccurs="unbounded" type="
                xes:ClassifierType"/>
            <xs:element name="trace" maxOccurs="unbounded" type="xes:TraceType"/>
          </xs:sequence>
          ...
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

...
```

1.7.3 Example

The example now declares three extensions: “Concept”, “Time”, and ”Organizational”. As a result of the first, the attribute with key “concept:name” now has the value that corresponds to the name of the activity related to this event. As a result of the second, the attribute with key “time:timestamp” now has the value that corresponds to the exact date and time the event occurred. As a result of the third, the attribute with key “org:resource” now has the value that corresponds to the resource that was involved in the event. Note that in earlier examples the attributes “Name”, “Timestamp”, and “Resource” were used, which of course did suggest some semantics, but which did not fix it. By using the extensions, the semantics has been fixed permanently.

```
<log xes.version="1.3" xes.features="nested-attributes" openxes.version="1.6" xmlns="
    http://www.xes-standard.org/">
  <extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept.
      xesext"/>
  <extension name="Time" prefix="time" uri="http://www.xes-standard.org/time.xesext"/>
  <extension name="Organizational" prefix="org" uri="http://www.xes-standard.org/org.
      xesext"/>
  <global scope="trace">
    <string key="concept:name" value="name"/>
  </global>
  <global scope="event">
    <string key="concept:name" value="Handle Email-Product Assistance"/>
    <string key="org:resource" value="resource"/>
    <date key="time:timestamp" value="2010-10-07T21:56:18.312+02:00"/>
    <string key="Operation" value="string"/>
    ...
    <string key="Service Type" value="string"/>
  </global>
```

```
  <classifier name="Operation" keys="Operation"/>
  <classifier name="Service Type" keys="Service Type"/>
  <classifier name="activity classifier" keys="Operation Service Type"/>
  <string key="Creator" value="Fluxicon Nitro"/>
  <trace>
    <string key="concept:name" value="Case1280"/>
    <string key="Creator" value="Fluxicon Nitro"/>
    <event>
      <string key="org:resource" value="FL"/>
      <date key="time:timestamp" value="2010-03-15T07:59:00.000+02:00"/>
      <string key="Operation" value="Handle Email"/>
      ...
    </event>
    ...
  </trace>
  <trace>
    ...
  </trace>
  ...
</log>
```

1.8 Standardized extensions

1.8.1 Concept extension

Extension prefix: `concept`

Extension URI: `http://www.xes-standard.org/concept.xesext`

Name

Attribute level: log, trace, event

Key : name

Type : Literal

Description: 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.

Instance

Attribute level: event

Key : name

Type : Literal

Description: The instance attribute is defined for events. It represents an identifier of the activity instance whose execution has generated the event.

1.8.2 Lifecycle extension

Extension prefix: `lifecycle`

Extension URI: `http://www.xes-standard.org/lifecycle.xesext`

Model

Attribute level: log

Key : model

Type : Literal

Description: 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.

Transition

Attribute level: event

Key : transition

Type : Literal

Description: 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.

1.8.3 Organizational extension

Extension prefix: `org`

Extension URI: `http://www.xes-standard.org/org.xesext`

Resource

Attribute level: event

Key : resource

Type : Literal

Description: The name, or identifier, of the resource having triggered the event.

Role

Attribute level: event

Key : role

Type : Literal

Description: The role of the resource having triggered the event, within the organizational structure.

Group

Attribute level: event

Key : group

Type : Literal

Description: The group within the organizational structure, of which the resource having triggered the event is a member.

1.8.4 Time extension

Extension prefix: `time`

Extension URI: `http://www.xes-standard.org/time.xesext`

Timetamp

Attribute level: event

Key : timestamp

Type : Timestamp

Description: The date and time, at which the event has occurred.

1.8.5 Semantic extension

Extension prefix: `semantic`

Extension URI: `http://www.xes-standard.org/semantic.xesext`

Model reference

Attribute level: log, trace, event, meta

Key : modelReference

Type : Literal

Description: References to model concepts in an ontology. Model references are stored in a literal string, as comma-separated URIs identifying the ontology concepts.

1.8.6 ID extension

Extension prefix: `identity`

Extension URI: `http://www.xes-standard.org/identity.xesext`

Id

Attribute level: log, trace, event, meta

Key : id

Type : ID

Description: The unique identity, encoded and implemented as UUID, of the log, trace, event, or parent attribute.

1.8.7 Cost extension

Extension prefix: `cost`

Extension URI: `http://www.xes-standard.org/cost.xesext`

Total

Attribute level: trace, event

Key : total

Type : Continuous

Description: Total cost incurred for a trace or an event. The value represents the sum of all the cost amounts within the element.

Currency

Attribute level: trace, event

Key : currency

Type : Literal

Description: The currency of all costs of this element in any valid currency format.

Amount

Attribute level: meta

Key : amount

Type : Continuous

Description: The value contains the cost amount for a cost driver.

Driver

Attribute level: meta

Key : total

Type : Literal

Description: The value contains the id for the cost driver used to calculate the cost.

Type

Attribute level: meta

Key : type

Type : Literal

Description: The value contains the cost type (e.g., Fixed, Overhead, Materials).

2 The OpenXES library

2.1 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.

2.2 XES Model Type Hierarchy

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

2.2.1 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.

2.2.2 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.

XAttribute 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.

- **XAttributeLiteral** The XAttributeLiteral interface extends the XAttribute interface with strongly typed methods to access and modify the string-based value.
- **XAttributeBoolean** The XAttributeBoolean interface extends the XAttribute interface with strongly typed methods to access and modify the boolean value.
- **XAttributeContinuous** The XAttributeContinuous interface extends the XAttribute interface with strongly typed methods to access and modify the double-precision floating point value.
- **XAttributeDiscrete** The XAttributeDiscrete interface extends the XAttribute interface with strongly typed methods to access and modify the long-precision integer value.
- **XAttributeTimestamp** 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).
- **XAttributeDuration** 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.
- **XAttributeID** The XAttributeID interface extends the XAttribute interface with strongly typed methods to access and modify the XID value.

XAttributeMap 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.

XAttributable The XAttributable interface defines capabilities of model type hierarchy elements, which can be equipped with attributes.

2.2.3 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/`).

XElement 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).

XLog 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.

XTrace 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.

XEvent 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.

2.2.4 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/`.

2.2.5 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.

2.3 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.

2.3.1 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 The class 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 The class 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.

XExtensionParser 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://www.xes-standard.org/`.

2.3.2 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
- XIdentityExtension
- XLifecycleExtension
- XOrganizationalExtension
- XSemanticExtension
- XTimeExtension
- XCostExtension

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.

2.4 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.

2.4.1 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`.

XEventClassifier 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.

XEventClass 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.

XEventClasses 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.

XEventAttributeClassifier 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.

Composite event classifiers 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.

2.4.2 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`.

XTimeBounds 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.

XAttributeInfo 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.

XLogInfo The XLogInfo interface defines a log info for OpenXES, and provides access to all aggregated information.

XLogInfoFactory 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.

2.4.3 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).

2.5 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.

2.5.1 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 "xes.gz" or ".xez" 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 "mxml.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.

2.5.2 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`.

XesSerializer 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.

XesXmlSerializer The XesXmlSerializer implements the serialization of XES models to the XML serialization, standardized earlier in this document.

XMxmlSerializer 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.

2.6 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.

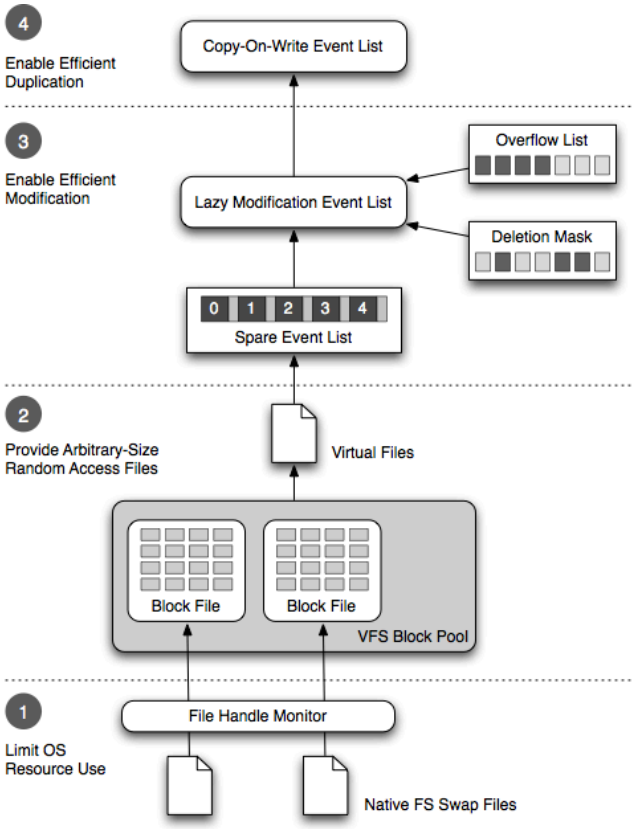

Figure 2.1: The architecture of disk-buffered event log data storage layer in OpenXES

2.6.1 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.

Figure 2.1 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.

Spare event list 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.

Lazy modification event list 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.

2.6.2 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.

2.7 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.

2.7.1 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.

2.7.2 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`.

2.7.3 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.

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">
  <!-- This file describes the XML serialization of the XES format for event log data.
      -->
  <!-- For more information about XES, visit http://www.xes-standard.org/ -->

  <!-- (c) 2012 by IEEE Task Force on Process Mining (http://www.win.tue.nl/ieetfpm) -->

  <!-- Date: June 12, 2012 -->
  <!-- Version: 1.1 -->
  <!-- Author: Christian Guenther (christian@fluxicom.com) -->
  <!-- Author: Eric Verbeek (h.m.w.verbeek@tue.nl) -->
  <!-- Change: Added AttributableType (list of attribute types now occurs only once) -->
  <!-- Change: Added id type -->
  <!-- Change: Made xes.features and openxes.version optional -->

  <!-- Date: November 25, 2009 -->
  <!-- Version: 1.0 -->
  <!-- Author: Christian Guenther (christian@fluxicom.com) -->

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

  <!-- Attributables -->
  <xs:complexType name="AttributableType">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="string" minOccurs="0" maxOccurs="unbounded" type="
          AttributeStringType"/>
      <xs:element name="date" minOccurs="0" maxOccurs="unbounded" type="
          AttributeDateType"/>
      <xs:element name="int" minOccurs="0" maxOccurs="unbounded" type="AttributeIntType"
          />
      <xs:element name="float" minOccurs="0" maxOccurs="unbounded" type="
          AttributeFloatType"/>
      <xs:element name="boolean" minOccurs="0" maxOccurs="unbounded" type="
          AttributeBooleanType"/>
      <xs:element name="id" minOccurs="0" maxOccurs="unbounded" type="AttributeIDType"/>
    </xs:choice>
  </xs:complexType>

  <!-- String attribute -->
  <xs:complexType name="AttributeStringType">
    <xs:complexContent>
      <xs:extension base="AttributeType">
            <xs:attribute name="value" use="required" type="xs:string"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- Date attribute -->
  <xs:complexType name="AttributeDateType">
    <xs:complexContent>
      <xs:extension base="AttributeType">
            <xs:attribute name="value" use="required" type="xs:dateTime"/>
```

```
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Integer attribute -->
<xs:complexType name="AttributeIntType">
  <xs:complexContent>
    <xs:extension base="AttributeType">
          <xs:attribute name="value" use="required" type="xs:long"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Floating-point attribute -->
<xs:complexType name="AttributeFloatType">
  <xs:complexContent>
    <xs:extension base="AttributeType">
          <xs:attribute name="value" use="required" type="xs:double"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Boolean attribute -->
<xs:complexType name="AttributeBooleanType">
  <xs:complexContent>
    <xs:extension base="AttributeType">
          <xs:attribute name="value" use="required" type="xs:boolean"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- ID attribute -->
<xs:complexType name="AttributeIDType">
  <xs:complexContent>
    <xs:extension base="AttributeType">
          <xs:attribute name="value" use="required" type="xs:string"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Extension definition -->
<xs:complexType name="ExtensionType">
  <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>

<!-- Globals definition -->
<xs:complexType name="GlobalsType">
  <xs:complexContent>
    <xs:extension base="AttributableType">
      <xs:attribute name="scope" type="xs:NCName" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Classifier definition -->
<xs:complexType name="ClassifierType">
  <xs:attribute name="name" type="xs:NCName" use="required"/>
  <xs:attribute name="keys" type="xs:token" use="required"/>
</xs:complexType>

<!-- Attribute -->
<xs:complexType name="AttributeType">
  <xs:complexContent>
    <xs:extension base="AttributableType">
          <xs:attribute name="key" use="required" type="xs:token"/>
```

```
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- Elements may contain attributes -->
  <xs:complexType name="ElementType">
    <xs:complexContent>
      <xs:extension base="AttributableType"/>
    </xs:complexContent>
  </xs:complexType>

  <!-- Logs are elements that may contain traces -->
  <xs:complexType name="LogType">
    <xs:complexContent>
      <xs:extension base="ElementType">
            <xs:sequence>
              <xs:element name="extension" minOccurs="0" maxOccurs="unbounded" type="
                  ExtensionType"/>
              <xs:element name="global" minOccurs="0" maxOccurs="2" type="GlobalsType"/>
              <xs:element name="classifier" minOccurs="0" maxOccurs="unbounded" type="
                  ClassifierType"/>
              <xs:element name="trace" maxOccurs="unbounded" type="TraceType"/>
            </xs:sequence>
            <xs:attribute name="xes.version" type="xs:decimal" use="required"/>
            <xs:attribute name="xes.features" type="xs:token"/>
            <xs:attribute name="openxes.version" type="xs:string"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- Traces are elements that may contain events -->
  <xs:complexType name="TraceType">
    <xs:complexContent>
      <xs:extension base="ElementType">
            <xs:sequence>
              <xs:element name="event" maxOccurs="unbounded" type="EventType"/>
            </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- Events are elements -->
  <xs:complexType name="EventType">
    <xs:complexContent>
      <xs:extension base="ElementType">
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

</xs:schema>
```

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 event
       log data. -->
  <!-- For more information about XES, visit http://www.xes-standard.org/ -->

  <!-- (c) 2012 IEEE Task Force on Process Mining (http://wwww.win.tue.nl/ieeetfpm) -->

  <!-- Date: June 12, 2012 -->
  <!-- Version: 1.1 -->
  <!-- Author: Christian Guenther (christian@fluxicom.com) -->
  <!-- Author: Eric Verbeek (h.m.w.verbeek@tue.nl) -->
  <!-- Change: Added AttributableType (list of attribute types now occurs only once) -->
  <!-- Change: Added id type -->

  <!-- Date: November 25, 2009 -->
  <!-- Version: 1.0 -->
  <!-- Author: Christian Guenther (christian@fluxicom.com) -->

  <!-- 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 name="log" minOccurs="0" maxOccurs="1" type="AttributableType"/>
        <xs:element name="trace" minOccurs="0" maxOccurs="1" type="AttributableType"/>
        <xs:element name="event" minOccurs="0" maxOccurs="1" type="AttributableType"/>
        <xs:element name="meta" minOccurs="0" maxOccurs="1" type="AttributableType"/>
      </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>

  <!-- Attributes -->
  <xs:complexType name="AttributableType">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="string" type="AttributeType"/>
      <xs:element name="date" type="AttributeType"/>
      <xs:element name="int" type="AttributeType"/>
      <xs:element name="float" type="AttributeType"/>
```

```
      <xs:element name="boolean" type="AttributeType"/>
      <xs:element name="id" type="AttributeType"/>
    </xs:choice>
  </xs:complexType>

  <!-- Attribute -->
  <xs:complexType name="AttributeType">
    <xs:sequence>
      <xs:element name="alias" minOccurs="0" maxOccurs="unbounded" type="AliasType"/>
    </xs:sequence>
    <xs:attribute name="key" use="required" type="xs:Name"/>
  </xs:complexType>

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

</xs:schema>
```

C Changes

C.1 Version 1.8

Revision 6976.

Eric Verbeek Changed attribute key from xs:Name to xs:token.

C.2 Version 1.7

Eric Verbeek Added cost extension.