EDXML SDK Documentation

Release 3.0.0

D.H.J. Takken

Mar 30, 2023

Contents

This package offers a Python implementation of the EDXML specification and various tools for EDXML application development. It can be installed using Pip:

```
pip install edxml
```

CHAPTER 1

Overview

The EDXML SDK can be used for generating and processing EDXML data. Additionally it offers some helpful tools to ease application development and unit testing. For example, it can generate visualizations of your ontology, show you how your EDXML templates will behave and normalize input data. Finally, the SDK contains a basic concept mining implementation.

CHAPTER 2

EDXML Data Modelling

2.1 Introduction to EDXML Data Modelling

EDXML is about transforming data into stories. Stories that both humans and machines can understand and reason about. This enables machines to read data much in the same way as a human being reads a novel, to learn while reading. Learning what the story is about and learning more and more as one paragraph is followed by another. Check https://edxml.org to learn more about this.

Modelling data in EDXML is the process of realizing what story the data tells and casting that into a model. EDXML has no predefined model. It provides the means to define whatever model fits your data best. This introduction will show what the process of data modelling looks like.

2.1.1 Understand Your Data

EDXML data is represented by means of *events*. Each event represents a little story, like a paragraph in a novel. An event can represent many different things, like a database record, a document, an e-mail message or a financial transaction. What a particular event represents is determined by its event type.

One of the first questions to be answered when modelling data as EDXML is: How can the original data be broken down into events? To answer that question, the analogy of events as paragraphs in a novel is very helpful. A single paragraph never tells a full story. It contributes to a story by conveying some bits of information in a structured way. Where a paragraph has a certain grammar, an event has a certain event type. Where a paragraph consists of words, an event consists of properties.

A typical event type may consists of about a dozen properties. When an event type gets much larger than that, it may be trying to tell too much and modelling it properly becomes more complicated. Many data sources already have a data structure that translates naturally into events, such as a row from a database table. Others may require splitting data records into multiple events.

The story of an event is pretty much the answer to the question:

> *What does this data mean, exactly?*

In this text we use server logs as an example. We will look into representing logging data from a file server that logs the FTP commands issued by users. We will assume that the log messages have already been parsed into JSON records, one JSON record per message. This might be achieved using some log message processor like LogStash, but this does not really matter for now.

We will assume that each command issued to the server generates one logging message. The question of how to break down the source data into EDXML events is therefore not a very complicated one: We will simply regard the logging messages as the paragraphs that tell the story of our FTP server. This implies that each logging message will result in one EDXML event.

Now suppose that the JSON records are structured like this:

```
{
  "time":    2016-10-11T21:04:36.167,
  "source":  "/var/log/ftp/command.4234.log",
  "offset":  37873,
  "server":  "192.168.1.20",
  "client":  "192.168.10.43",
  "user":    "alice",
  "command": "quit"
}
```

What does this data mean, exactly? In order to answer that question, we need to do a bit of homework. For example, what is the time zone of the `time` field in the JSON data? And what is that `offset` field all about? Is that `server` field always an IP address or can it also contain a host name?

For our example case, we will assume that:

- the `time` field is a time specified in UTC
- the `offset` and `source` fields represent the offset in the original logging file, added by the log forwarder
- the `server` and `client` fields are always IPv4 addresses

2.1.2 The Event Story

Now that we understand the data, let us write down the exact meaning of the above JSON record, in plain english:

> *On October 11th 2016 at 21:04:36.167h (UTC), a user named 'alice' issued command 'quit' on FTP server 192.168.1.20. The command was issued from a device having IP address 192.168.10.43.*

The above text is called an *event story*. The event story is the very foundation of your data model and writing it down is the most important step in EDXML data modelling. It forces you to think about the exact meaning and significance of your data. In a minute, you will see how this text naturally results in a data model.

There is more to say about the JSON record, but we will get to that later. At this point, we know the story we want to tell and we verified that all the required information is available in the original data.

2.1.3 The Story Template

Event stories like the one above are closely related to *story templates* in EDXML. For this reason, we will use story templates as the first step towards an EDXML data model for our FTP events. Story templates transform the various event properties into human readable text. The resulting text explains the exact meaning of the event data, in terms of the event properties. Especially if you are just getting started with EDXML data modelling, it is advisable to begin a modelling task by writing down the story template that yields event stories like the one above. Once a satisfactory template has been established, this will also result in a list of required properties, as these are part of the template. Let us convert the above event story into a story template by replacing the variable parts with place holders:

> *On [[time]], a user named '[[user]]' issued command '[[command]]' on FTP server [[server]]. The command was issued from a device having IP address [[client]].*

In the above template, we used event properties that have exactly the same names as the fields in the JSON record. We can do that in this case because the JSON records are flat, like EDXML events, and the JSON field names just happen to be valid EDXML property names.

In order to convert the date into something slightly more human friendly, we can use a *string formatter*. Let us use the `date_time` formatter to display the time rounded to the nearest second:

> *On [[date_time:time,second]], a user named '[[user]]' issued command '[[command]]' on FTP server [[server]]. The command was issued from a device having IP address [[client]].*

Note: The EDXML specification lists a number of string formatters that you can use.

In our example, we assume that all JSON fields are present in every input JSON record. This makes writing down a template quite straight forward. EDXML story templates also supports creating dynamic event stories that change depending on which fields have a value and which do not. Refer to the full specification for details.

2.1.4 The Event Type

At this point, we have our event story, a story template and we know which properties our EDXML event type will have. It is time to actually define our first event type! We will do so by using the *`EventTypeFactory`* class. This provides a declarative way to define event types and allows generating a full EDXML ontology from it. Let us begin writing our event type definition, adding just the `time` property for now:

```
from edxml.ontology import EventTypeFactory
from edxml_bricks.generic import GenericBrick


class FtpTypeFactory(EventTypeFactory):

    TYPES = ['org.myorganization.logs.ftp']

    TYPE_PROPERTIES = {
        'org.myorganization.logs.ftp': {
            'time': GenericBrick.OBJECT_DATETIME
        }
    }
```

The EventTypeFactory class offers some constants that we can populate. The ontology will be generated from these constants. The *`TYPES`* constant simply lists the names of the event types that we want to define. The *`TYPE_PROPERTIES`* constant lists the properties for each event type. It is a dictionary mapping property names to the *object type* of the values that we will store in the property. In this case we use `GenericBrick.OBJECT_DATETIME`.

The `GenericBrick.OBJECT_DATETIME` we use here refers to the definition of an object type from an *ontology brick*. An ontology brick is a reusable component for building ontologies. By using this object type in stead of defining our own we make sure that our ontology defines time in the same way as other ontologies that also use this ontology brick. This is important, as it makes our ontology compatible with ontologies defined by other EDXML data sources.

Note: The EDXML Foundation maintains a collection of ontology bricks for this purpose. The definition of the object type we use in our example can be found here.

Now, if we want to generate an actual EDXML document containing our ontology, we can use an EDXML writer to do so:

```
from edxml import EDXMLWriter

ontology = FtpTypeFactory().generate_ontology()

EDXMLWriter().add_ontology(ontology).close()
```

By default the EDXML writer will write the resulting document to standard output. The ontology that we defined so far should result in an EDXML document similar to the one shown below:

```
<?xml version='1.0' encoding='utf-8'?>
<edxml xmlns="http://edxml.org/edxml" version="3.0.0">
<ontology xmlns="http://edxml.org/edxml">
  <object-types>
    <object-type name="datetime"
                 display-name-singular="time stamp"
                 display-name-plural="time stamps"
                 description="a date and time in ISO 8601 format"
                 data-type="datetime"
                 version="1"/>
  </object-types>
  <concepts/>
  <event-types>
    <event-type name="org.myorganization.logs.ftp"
                display-name-singular="org myorganization logs ftp"
                display-name-plural="org myorganization logs ftps"
                description="org.myorganization.logs.ftp"
                summary="no description available"
                story="no description available"
                version="1">
      <properties>
        <property name="time"
                  object-type="datetime"
                  description="time"
                  optional="false"
                  multivalued="false"
                  confidence="10"/>
      </properties>
    </event-type>
  </event-types>
  <sources/>
</ontology>
</edxml>
```

As you can see the document contains just our event type definition, there is no event data yet. Note that the event type contains a lot more details than we specified. Most of these are defaults of the EDXML implementation in the SDK. Others, like the display names, are brave attempts to guess them based on other information. All of these details can be adjusted by using the constants of the *EventTypeFactory* class.

Let us extend our event type definition a bit more. We can use the *TYPE_STORIES* constant to set our story template. And, while we are at it, we also add the remaining properties:

```
from edxml.ontology import EventTypeFactory

from edxml_bricks.generic import GenericBrick
from edxml_bricks.computing.generic import ComputingBrick
```

(continues on next page)

(continued from previous page)

```
from edxml_bricks.computing.networking.generic import NetworkingBrick


class FtpTypeFactory(EventTypeFactory):

    TYPES = ['org.myorganization.logs.ftp']

    TYPE_PROPERTIES = {
        'org.myorganization.logs.ftp': {
            'time': GenericBrick.OBJECT_DATETIME,
            'user': ComputingBrick.OBJECT_USER_NAME,
            'command': GenericBrick.OBJECT_STRING_UTF8,
            'server': NetworkingBrick.OBJECT_HOST_IPV4,
            'client': NetworkingBrick.OBJECT_HOST_IPV4,
        }
    }

    TYPE_STORIES = {
        'org.myorganization.logs.ftp':
            'On [[date_time:time,second]], a user named "[[user]]" issued '
            'command "[[command]]" on FTP server [[server]]. The command '
            'was issued from a device having IP address [[client]].'
    }
```

Again we used some object types from the public shared brick collection. Our event type also contains one property, `command`, for which the collection does not offer a fitting object type. For the time being we will use some generic string object type.

2.1.5 Event Uniqueness

Each EDXML event is uniquely identified by its hash. This hash is computed from its event type, event source and its *hashable properties*. By default none of the properties of an event is hashable. That won't work well because we will produce many different variants of one and the same event. We really need to think about choosing hashable properties.

The question we need to answer now is: What makes an event of our event type truly unique? Of course we could just mark all properties as hashable. But what if a user manages to run the same FTP command multiple times, within the time resolution of the time stamps in the log? Then all of these events will still have the same hash. Machines will consider them as duplicates of a single event. If we want our events to accurately represent the original data, we need to come up with something better.

In the original JSON records we find the name of the log file and the offset of the log message in that file. Let us assume that this combination uniquely identifies a log message. Then we can extend our event type by adding two properties and specify both as hashable:

```
from edxml.ontology import EventTypeFactory

from edxml_bricks.generic import GenericBrick
from edxml_bricks.computing.files import FilesBrick
from edxml_bricks.computing.generic import ComputingBrick
from edxml_bricks.computing.networking.generic import NetworkingBrick


class FtpTypeFactory(EventTypeFactory):

```

(continues on next page)

(continued from previous page)

```
    TYPES = ['org.myorganization.logs.ftp']

    TYPE_PROPERTIES = {
        'org.myorganization.logs.ftp': {
            'time': GenericBrick.OBJECT_DATETIME,
            'user': ComputingBrick.OBJECT_USER_NAME,
            'command': GenericBrick.OBJECT_STRING_UTF8,
            'server': NetworkingBrick.OBJECT_HOST_IPV4,
            'client': NetworkingBrick.OBJECT_HOST_IPV4,
            'file': FilesBrick.OBJECT_FILE_PATH,
            'offset': GenericBrick.OBJECT_SEQUENCE,
        }
    }

    TYPE_STORIES = {
        'org.myorganization.logs.ftp':
            'On [[date_time:time,second]], a user named "[[user]]" issued '
            'command "[[command]]" on FTP server [[server]]. The command '
            'was issued from a device having IP address [[client]] and was'
            'originally logged in [[file]] at offset [[offset]].'
    }

    TYPE_HASHED_PROPERTIES = {
        'org.myorganization.logs.ftp': ['file', 'offset']
    }
```

Note: Event hashing is explained in more detail in *Event Uniqueness & Hashing*.

2.1.6 Event Order

The order of events as they appear in an EDXML document has no significance. The event order must be determined by comparing their event properties. When we discussed uniqueness we touched on a potential problem here: The time resolution may not be sufficient to differentiate between log messages produced in quick succession. Not only does this pose a challenge to producing unique events, it also means that the original log message ordering may get lost in translation.

As detailed in the specification the logical event order is determined by comparing time spans and, if that is not sufficient, by subsequently comparing sequence numbers if the event type defines these. In our event type the `offset` property looks like a nice candidate for a sequence number. Let us put it to use right away:

```
from edxml.ontology import EventTypeFactory

from edxml_bricks.generic import GenericBrick
from edxml_bricks.computing.files import FilesBrick
from edxml_bricks.computing.generic import ComputingBrick
from edxml_bricks.computing.networking.generic import NetworkingBrick


class FtpTypeFactory(EventTypeFactory):

    TYPES = ['org.myorganization.logs.ftp']

    TYPE_PROPERTIES = {
```

(continues on next page)

(continued from previous page)

```
        'org.myorganization.logs.ftp': {
            'time': GenericBrick.OBJECT_DATETIME,
            'user': ComputingBrick.OBJECT_USER_NAME,
            'command': GenericBrick.OBJECT_STRING_UTF8,
            'server': NetworkingBrick.OBJECT_HOST_IPV4,
            'client': NetworkingBrick.OBJECT_HOST_IPV4,
            'file': FilesBrick.OBJECT_FILE_PATH,
            'offset': GenericBrick.OBJECT_SEQUENCE,
        }
    }

    TYPE_STORIES = {
        'org.myorganization.logs.ftp':
            'On [[date_time:time,second]], a user named "[[user]]" issued '
            'command "[[command]]" on FTP server [[server]]. The command '
            'was issued from a device having IP address [[client]] and was'
            'originally logged in [[file]] at offset [[offset]].'
    }

    TYPE_HASHED_PROPERTIES = {
        'org.myorganization.logs.ftp': ['file', 'offset']
    }

    TYPE_SEQUENCES = {
        'org.myorganization.logs.ftp': 'offset'
    }
```

2.1.7 Concepts

Concepts unlock some of the most powerful features of EDXML, most notably concept mining. Associating properties with concepts and mutually relating those concepts makes your data reach its full potential.

Let us once again ask ourselves a question: Which event properties are identifiers of some concept? To answer that question, we remind ourselves of the analogy of concepts being like the characters in a novel. Concepts are the things that the story is about. The heroes in our story are two computers and a user. Now let us add this information to the event type:

```
TYPE_PROPERTY_CONCEPTS = {
    'org.myorganization.logs.ftp': {
        'user': {ComputingBrick.CONCEPT_USER_ACCOUNT: 7},
        'server': {ComputingBrick.CONCEPT_COMPUTER: 8},
        'client': {ComputingBrick.CONCEPT_COMPUTER: 8},
    }
}
```

Here we associate event type properties with concepts. And once again we use some concept definitions provided by the public ontology brick collection. In case you are curious what the definition of a computer looks like, you can find it here.

The integer numbers that you see displayed for each concept are *concept confidences*. For the precise definition we refer to the EDXML specification. Long story short: These values are rough indicators of how strong of an identifier the property is for the concept that it is associated with.

The reasoning for choosing the specific values displayed above is as follows. The name of a user account is generally not a very strong identifier because many different unrelated accounts on different computers may share the same

account name, like "root" or "admin". We consider the IPv4 address of a computer to be a somewhat stronger identifier because multiple computers having the same IP address is less common.

Choosing confidences is not an exact science. The values are used by concept mining algorithms to *estimate* the confidence of their output.

The next thing to consider is how these concepts relate to one another:

- The user makes use of both the server and the client computer.
- The server serves the client computer

Concept relations are somewhat more complex to express than most aspects of event types. For that reason there are no class constants that we can populate to define them. We need to use the *EDXML Ontologies* API for this. The event type factory class uses the *create_event_type()* method to generate its event types. By overriding it we can adjust the event type definitions to our liking. Let us define the relation between the server and client computers:

```
    @classmethod
    def create_event_type(cls, event_type_name, ontology):
        ftp = super().create_event_type(event_type_name, ontology)

        ftp['server'].relate_inter('serves', 'client').because(
            'FTP logs show [[client]] executing commands on [[server]]')
```

Here we use an *inter-concept* relation, which indicates that the relation relates two distinct computers rather than linking information about one and the same computer. The relation enables concept mining algorithms to use FTP logs to discover networks of interconnected computers. And, because we used shared ontology bricks, the FTP log events are easily combined with other EDXML data sources which tell us something about the structure of the computer network, about the computers themselves or about their users.

Note that the relation definition also makes use of an EDXML template. This template can be used to translate reasoning steps into English text.

2.1.8 Event Attachments

An event type can specify that its events may have attachments. Attachments can be used to supply additional information that supports the story. An attachment can contain many things, ranging from a plain text string to a Base64 encoded picture.

There are some considerations to be made when deciding to store something in a property or in an attachment. As properties are typically part of the event story, long strings or binary strings do not help to keep the event story clear and concise. Attachments do not play any role in concept mining or correlating events. If that is no issue, storing a value as an attachment should be fine.

Let us use an attachment to store the original JSON record along with its resulting event:

```
    TYPE_ATTACHMENTS = {
        'org.myorganization.logs.ftp': ['original']
    }

    TYPE_ATTACHMENT_DISPLAY_NAMES = {
        'org.myorganization.logs.ftp': {'original': ['original JSON record']}
    }

    TYPE_ATTACHMENT_MEDIA_TYPES = {
        'org.myorganization.logs.ftp': {'original': 'application/json'}
    }
```

Here we define one attachment with attachment id `original` and specify its display name and media type. Note that there is another class constant that can be used to indicate that the attachments are Base64 encoded. In the case of a JSON record the default attachment encoding (plain text) is fine.

2.1.9 Ontology Evolution

As mentioned before, event types have many more aspects than the ones we have shown here. To keep example code short we left most of these at default values. Omitting details can also be done purposely when designing event types for real. Event types are versioned and *most* of their aspects can be updated later. The emphasis on *most* points us at a critically important step in ontology design:

> **Warning:** Make sure that you correctly specify all immutable aspects in version 1 of your event type.

The main issue is that some aspects of event type definitions cannot be changed after it has been used to generate events. Well, you could do that but that may yield two mutually conflicting event type definitions. The EDXML specification specifies how to update an event type that is guaranteed to work and will be accepted by any EDXML implementation. It also guarantees that events generated using previous versions of their event type are still valid according to newer versions.

To make that happen, some aspects of an event type definition are static. They cannot be changed in newer versions. It is crucial to get these aspects right from the start.

The specification details which aspects of an ontology can be updated and how. In practise, updating an event type involves outputting an ontology containing the updated event type while making sure that the version of the event type is incremented. The EventTypeFactory class has a *`class constant`* to set the event type version.

2.1.10 Selecting Object Types

For the `command` property we used a generic string object type. This would work just fine for generating events, it gets the job done. But is also has some consequences that we should consider.

The event type we just defined may not be the only event type using this object type. This may yield surprises when the events are combined with data from other sources that use the same object type for completely different kinds of data. Logged FTP commands may end up on a big pile of object values together with who knows what else.

When two EDXML events refer to the same object value of the same object type, they conceptually share a single object. Correlating events by finding events that share objects is a common thing to do in EDXML data analysis. When a single object can represent two wildly different things, machines may see links between events where there actually is none.

Long story short, it is a good idea to use a more specific object type for representing FTP commands. In doing so we have two options. We can define a *private* object type or a *public* object type. While there is no such thing as a private or public object type in the EDXML specification, it is a useful distinction to make.

A private object type is intended to be used by a single EDXML data source. Its name includes a namespace that identifies the data source to make sure that the chances of another data source defining the same object type is minimal. This also implies that object values can never be correlated with events from other data sources and that the objects are not useful for concept mining.

A public object type is the opposite. It is intended to be shared among multiple data sources. Its name does not refer to any specific data source. The object values are expected to correlate events from multiple event types or data sources in a meaningful way. And, ideally, it is shared by adding it to the public ontology bricks collection.

Suppose that we decide that finding the exact same FTP command in other event types does not really mean anything to us. Then a private object type is the way to go. We also do not have any concept associated with FTP commands, so we are not missing out on the concept mining end if we use a private object type.

The most convenient way to define our own private object type is to create our own ontology brick:

```
from edxml.ontology import Brick
from edxml.ontology import DataType


class MyOwnBrick(Brick):

    OBJECT_FTP_COMMAND = 'org.myorganization.ftp.command'

    @classmethod
    def generate_object_types(cls, target_ontology):

        yield target_ontology.create_object_type(cls.OBJECT_FTP_COMMAND)\
            .set_description('a command issued on an FTP server')\
            .set_data_type(DataType.string())\
            .set_display_name('FTP command')
```

We can use this brick in the exact same way as we did with the bricks from the public brick collection. We leave that as an exercise for the reader.

2.1.11 Event Sources

Every EDXML event has a source. An event source is an URI that provides the means to organize EDXML events in a virtual hierarchy. The position inside this hierarchy represents the origin of the data. For example, event sources may include the name of the data source, or the organization that owns the data. Event sources can also provide a convenient means to filter data. For example, the source might allow filtering data on a specific department within an organization. Including a date in the source URI may be very convenient for implementing data retention policies or efficiently keeping metrics like events per month / week / day.

> *Since EDXML source URIs from all data sources combined form a virtual hierarchical tree structure, it is advisable to establish an organization-wide convention for generating source URIs. Document it or, even better, implement it in a shared code library.*

EDXML source URIs should be properly namespaced to prevent any collisions with the URIs produced by another data source. This can be done by including the owner of the data in the URI, as illustrated below:

```
/org/myorganization/offices/amsterdam/logs/2021/q2/
```

Unlike object types and concepts, event sources are usually not shared between data sources. For that reason ontology bricks do not provide the means to define event sources. Defining event sources can be done using the *ontology API* or using a *transcoder mediator*.

EDXML requires outputting the ontology before outputting any events. As event sources are part of the ontology, this suggests that all event sources must be defined upfront. However, EDXML allows outputting ontology updates at any point while producing events, which means that sources can be defined dynamically.

> A full working version of the examples shown on this page can be found on Github.

2.2 Event Uniqueness & Hashing

Each EDXML event is uniquely identified by its hash. How this hash is computed is detailed in the EDXML specification. Two events may have differing properties while their hashes are identical. This is due to the fact that an event type may define to only include specific properties while computing its hash: The *hashable properties*.

When two physical events share the same hash, they are instances of one and the same logical event. The physical events are said to collide and can be merged together. This characteristic gives events a unique and persistent identifier. The event can be referred to by its hash even as it evolves over time.

Generating colliding events allows an event source to produce an update for a previously generated event. A recipient can merge the events to track the current state of some variable record in the source. A typical example is representing tickets from a ticketing system that have a variable status ('open', 'in progress', 'done').

Defining hashable properties can be done by using the *`make_hashed()`* method on the event property:

```
my_event_type.create_property('time', 'datetime').make_hashed()
```

Multiple instances of the same logical event can be merged using the various merge strategies defined in the EDXML specification. By default, the merge strategy of all properties is set to `any`. Setting a different merge strategy on an event property can be done using the various `merge_...()` methods on the property. A quick example:

```
my_event_type['time-end'].merge_max()
```

2.2.1 Choosing hashable properties

Choosing which properties to include in the hash is a critical step in event type design. It determines if and how events can evolve. It shows what makes an event unique. It has a role in defining *parent-child relations* between event types. Updating the set of hashable properties for a particular event type is not possible, you must get it right from the start. Well, you can do it but this yields two conflicting definitions of the same event type.

Some data sources already produce unique persistent identifiers for their data records like a UUID or a hash. In that case, you are lucky: All that needs to be done is storing this unique identifier in an event property and marking it as the only hashable property. Problem solved. However, there is another aspect of event uniqueness to consider here: Semantics. Using that UUID from the data source to uniquely identify your EDXML events will work perfectly, but the resulting events will not convey what it really is that makes them unique.

Let us clarify the semantics problem by looking at an example. Consider a data source yielding records that contain address information comprising of a postal code and a house number. The data source produces exactly one record for each unique address. Each record also contains a unique database record identifier. Modelling an event type to represent these records could be done by using the database identifiers provided by the source for uniqueness. However, the fact that the combination of postal code and house number makes each event unique will be lost. Data analysis algorithms can no longer infer what makes these events unique by inspecting the event type definition. By marking the two properties that contain the postal code and the house number as hashable in stead, the semantics of the event type are much stronger.

2.3 Modelling Hierarchical Structures

EDXML events have a simple, flat structure. While this flat structure has many advantages, it also poses some challenges. When modelling data from sources that produce nested data structures like JSON or XML, the original data records typically need to be broken up into multiple EDXML events. The original hierarchical structure gets lost in the process. As a result, it becomes difficult to grasp the relationships between the various types of events.

In EDXML, modelling of nested data structures is done by defining parent-child relations between event types. These definitions enable machines to aggregate multiple events into hierarchical structures as desired.

Before we look at how we can use the SDK to define parent / child relationships, let us recap the EDXML specification on the subject. Parent and child events have the following characteristics:

- A parent event can identify all of its children in terms of its own properties
- A child event can identify its parent in terms of its own properties
- A child event has at most one parent
- The parent-child relationship cannot be broken by updating parent or child

This is only possible when:

1. The parent event type defines at least one hashed property
2. The child event contains a copy of all hashed properties of the parent
3. The properties that the child shares with the parent cannot be updated

As an example, suppose that we wish to model a directory of files stored on a computer. We define one event type `dir` describing the directory itself, which may contain a hashed property `name` containing the directory name and possibly some aggregate information like the total number of files it contains. Then we define another event type `file` describing a file contained in the directory. The file event type contains a property `dir` containing the name of the directory that contains the file.

> *Refer to this document to learn about defining hashed properties*

2.3.1 Defining parent / child relations

Now we can express the parent-child relationship between both event types as follows:

```
file.make_child('contained in', dir.make_parent('containing', file)).map('dir', 'name
↪')
```

The API to define the relationship is defined to yield code that is actually quite readable. In plain English, it says:

> *Make a file into a child contained in the directory, which is the parent containing the file, by mapping the directory of the file to the name of the directory.*

Note that the text fragments 'contained in' and 'containing' are more than just syntactic sugar. These texts are in fact the parent description and siblings description that the EDXML specification is talking about. The call to *`make_child()`* returns an instance of the *`edxml.ontology.EventTypeParent`* class. The call to *`map()`* creates the property map. As detailed in the EDXML specification, the property map defines which object values are shared between the parent and child events and which property of the child corresponds with each of the properties of the parent event.

> *It may be convenient to match the names of the properties that bind the parent and child event types. In that case, the second argument to the map() method can be omitted.*

Parent / child relations can also be defined in a declarative way when using an *event type factory* or a *transcoder*. The following example illustrates what this might look like:

```
from edxml.ontology import EventTypeFactory


class FileSystemTypes(EventTypeFactory):
    TYPES = ['dir', 'file']
    TYPE_PROPERTIES = {
```

(continues on next page)

(continued from previous page)

```
        'dir': {'name': 'filesystem-name'},
        'file': {'dir': 'filesystem-name'}
    }
    TYPE_HASHED_PROPERTIES = {
        'dir': ['name'],
        'file': ['dir']
    }
    PARENT_MAPPINGS = {
        'file': {'dir': 'name'}
    }
    PARENTS_CHILDREN = [
        ['dir', 'containing', 'file']
    ]

    CHILDREN_SIBLINGS = [
        ['file', 'contained in', 'dir']
    ]
```

CHAPTER 3

API Documentation

3.1 Writing EDXML Data

The EDXML SDK features several components for producing EDXML data, all based on the excellent lxml library. Data generation is incremental, which allows for developing efficient system components that generate or process EDXML data in a streaming fashion.

3.1.1 EDXMLWriter

The *EDXMLWriter* class is the prime, low level EDXML generator. For most practical use cases a *transcoder* offers a superior means for generating EDXML data.

Using the EDXML writer is pretty straight forward, as the following example demonstrates:

```
from edxml import EDXMLWriter, EDXMLEvent
from edxml.ontology import Ontology

# Create basic ontology
ontology = Ontology()
ontology.create_object_type(name='some.object.type')
source = ontology.create_event_source(uri='/some/source/')
event_type = ontology.create_event_type(name='some.event.type')
event_type.create_property(name='prop', object_type_name='some.object.type')

# Create an EDXML event
event = EDXMLEvent(
    properties={'prop': {'FooBar'}},
    event_type_name=event_type.get_name(),
    source_uri=source.get_uri()
)

# Generate EDXML data (writes to stdout)
with EDXMLWriter() as writer:
```

(continues on next page)

(continued from previous page)

```
    writer.add_ontology(ontology)
    writer.add_event(event)
```

Class Documentation

class edxml.**EDXMLWriter**(*output=<_io.BufferedWriter name='<stdout>'>*, *validate=True*, *log_repaired_events=False*, *pretty_print=True*)

Bases: `object`

Class for generating EDXML streams

The output parameter is a file-like object that will be used to send the XML data to. When the output parameter is set to None, the generated XML data will be returned by the methods that generate output.

The optional validate parameter controls if the generated EDXML stream should be auto-validated or not. Automatic validation is enabled by default.

Parameters

- **output** (*file*) – File-like output object
- **validate** (*bool*) – Enable output validation (True) or not (False)
- **log_repaired_events** (*bool*) – Log repaired events (True) or not (False)

enable_auto_repair_normalize(*event_type_name*, *property_names*)

Enables automatic repair of the property values of events of specified type. Whenever an invalid event is generated by the mediator it will try to repair the event by normalizing object values of specified properties.

Parameters

- **event_type_name** (*str*) –
- **property_names** (*List[str]*) –

Returns The EDXMLWriter instance

Return type *edxml.EDXMLWriter*

enable_auto_repair_drop(*event_type_name*, *property_names*)

Allows dropping invalid object values from the specified event properties while repairing invalid events. This will only be done as a last resort when normalizing object values failed or is disabled.

Parameters

- **event_type_name** (*str*) –
- **property_names** (*List[str]*) –

Returns The EDXMLWriter instance

Return type *edxml.EDXMLWriter*

ignore_invalid_events(*warn=False*)

Instructs the EDXML writer to ignore invalid events. After calling this method, any event that fails to validate will be dropped. If warn is set to True, a detailed warning will be printed, allowing the source and cause of the problem to be determined.

Note: This has no effect when event validation is disabled.

Parameters **warn** (*bool*) – Print warnings or not

Returns The EDXMLWriter instance

Return type *EDXMLWriter*

flush()
When no output was provided when creating the EDXML writer, any generated EDXML data is stored in an internal buffer. In that case, this method will return the content of the buffer and clear it. Otherwise, an empty string is returned.

Returns Generated EDXML data

Return type bytes

add_ontology(*ontology*)
Writes an EDXML ontology element into the output.

Parameters **ontology** (edxml.ontology.Ontology) – The ontology

Returns The EDXMLWriter instance

Return type edxml.writer.EDXMLWriter

close()
Finalizes the output data stream.

Returns The EDXMLWriter instance

Return type edxml.writer.EDXMLWriter

add_event(*event*, *sort=False*)
Adds specified event to the output data stream.

When the sort parameter is set to True, the properties, attachments and event parents are sorted as required for obtaining the event in its normal form as defined in the EDXML specification. While this does not actually output the events in their normal form, the sorting does make it easier to spot relevant differences between events.

Parameters

- **event** (edxml.EDXMLEvent) – The event
- **sort** (*bool*) – Sort event components yes or no

Returns The EDXMLWriter instance

Return type edxml.writer.EDXMLWriter

add_foreign_element(*element*)
Adds specified foreign element to the output data stream.

Parameters **element** (*etree._Element*) – The element

Returns The EDXMLWriter instance

Return type edxml.writer.EDXMLWriter

3.2 Reading EDXML Data

The EDXML SDK features several classes and subpackages for parsing EDXML data streams, all based on the excellent lxml library. All EDXML parsers are incremental, which allows for developing efficient system components that process a never ending stream of input events.

3.2.1 EDXML Parsers

All EDXML parsers are based on the *EDXMLParserBase* class, which has several subclasses for specific purposes. During parsing, the parser generates calls to a set of callback methods which can be overridden to process input data. There are callbacks for processing events and tracking ontology updates.

The *EDXMLParserBase* class has two types of subclasses, *push parsers* and *pull parsers*. Pull parsers read from a provided file-like object in a blocking fashion. Push parsers need to be actively fed with string data. Push parsers provide control over the input process, which allows implementing efficient low latency event processing components.

The two most used EDXML parsers are *EDXMLPullParser* and *EDXMLPushParser*. For the specific purpose of extracting ontology data from EDXML data, there are *EDXMLOntologyPullParser* and *EDXMLOntologyPushParser*. The latter pair of classes skip event data and only invoke callbacks when ontology information is received.

An example of using the pull parser is shown below:

```
import os

from edxml import EDXMLPullParser


class MyParser(EDXMLPullParser):
    def _parsed_event(self, event):
        # Do whatever you want here
        ...

    def _parsed_ontology(self, ontology):
        # Do whatever you want here
        ...


with MyParser() as parser:
    parser.parse(os.path.dirname(__file__) + '/input.edxml')
```

Besides extending a parser class and overriding callbacks, there is secondary mechanism specifically for processing events. EDXML parsers allow callbacks to be registered for specific event types or events from specific sources. These callbacks can be any Python callable. This allows EDXML data streams to be processed using a set of classes, each of which registered with the parser to process specific event data. The parser takes care of routing the events to the appropriate class.

3.2.2 EventCollection

In stead of reading EDXML data in a streaming fashion it can also be useful to read and access an EDXML document as a whole. This can be done using the *EventCollection* class. The following example illustrates this:

```
import os

from edxml import EventCollection

data = open(os.path.dirname(__file__) + '/input.edxml', 'rb').read()

collection = EventCollection.from_edxml(data)

for event in collection:
    print(event)
```

3.2.3 Class Documentation

The class documentation of the various parsers can be found below.

- *EDXMLParserBase*
- *EDXMLPushParser*
- *EDXMLPullParser*
- *EDXMLOntologyPushParser*
- *EDXMLOntologyPullParser*

EDXMLParserBase

class edxml.**EDXMLParserBase**(*validate=True*)
Bases: object

This is the base class for all EDXML parsers.

Create a new EDXML parser. By default, the parser validates the input. Validation can be disabled by setting validate = False

Parameters **validate** (*bool, optional*) – Validate input or not

close()
Close the parser after parsing has finished. After closing, the parser instance can be reused for parsing another EDXML data file.

Returns The EDXML parser

Return type *EDXMLParserBase*

get_event_counter()
Returns the number of parsed events. This counter is incremented after the _parsedEvent callback returned.

Returns The number of parsed events

Return type int

get_event_type_counter(*event_type_name*)
Returns the number of parsed events of the specified event type. These counters are incremented after the _parsedEvent callback returned.

Parameters **event_type_name** (*str*) – The type of parsed events

Returns The number of parsed events

Return type int

get_ontology()
Returns the ontology that was read by the parser. The ontology is updated whenever new ontology information is parsed from the input data.

Returns The parsed ontology

Return type *edxml.ontology.Ontology*

set_custom_event_class(*event_class*)
By default, EDXML parsers will generate ParsedEvent instances for representing event elements. When this method is used to set a custom element class, this class will be instantiated in stead of ParsedEvent. This can be used to implement custom APIs on top of the EDXML events that are generated by the parser.

Note: In is strongly recommended to extend the ParsedEvent class and implement additional class methods on top of it.

Note: Implementing a custom element class that can replace the standard etree.Element class is tricky, be sure to read the lxml documentation about custom Element classes.

Parameters **event_class** (*etree.ElementBase*) – The custom element class

Returns The EDXML parser

Return type *EDXMLParserBase*

set_event_source_handler(*source_patterns*, *handler*)

Register a handler for specified event sources. Whenever an event is parsed that has an event source URI matching any of the specified regular expressions, the supplied handler will be called with the event (which will be a ParsedEvent instance) as its only argument.

Multiple handlers can be installed for a given event source, they will be invoked in the order of registration. Event source handlers are invoked after event type handlers.

Parameters

- **source_patterns** (*List[str]*) – List of regular expressions
- **handler** (*callable*) – Handler

Returns The EDXML parser

Return type *EDXMLParserBase*

set_event_type_handler(*event_types*, *handler*)

Register a handler for specified event types. Whenever an event is parsed of any of the specified types, the supplied handler will be called with the event (which will be a ParsedEvent instance) as its only argument.

Multiple handlers can be installed for a given type of event, they will be invoked in the order of registration. Event type handlers are invoked before event source handlers.

Parameters

- **event_types** (*List[str]*) – List of event type names
- **handler** (*callable*) – Handler

Returns The EDXML parser

Return type *EDXMLParserBase*

EDXMLPushParser

class edxml.**EDXMLPushParser**(*validate=True*, *foreign_element_tags=None*)

Bases: edxml.parser.EDXMLParserBase

An incremental push parser for EDXML data. Unlike the pull parser, this parser does not read data by itself and does not block when the data stream dries up. It needs to be actively fed with stings, allowing full control of the input process.

Optionally, a list of tags of foreign elements can be supplied. The tags must prepend the namespace in James Clark notation. Example:

['{http://some/foreign/namespace}attribute']

These elements will be passed to the _parse_foreign_element() when encountered.

Note: This class extends EDXMLParserBase, refer to that class for more details about the EDXML parsing interface.

---

feed(*data*)

Feeds the specified string to the parser. A call to the feed() method may or may not trigger calls to callback methods, depending on the size and content of the passed string buffer.

Parameters **data** (*bytes*) – String data

EDXMLPullParser

class edxml.**EDXMLPullParser**(*validate=True*)

Bases: edxml.parser.EDXMLParserBase

An blocking, incremental pull parser for EDXML data, for parsing EDXML data from file-like objects.

Note: This class extends EDXMLParserBase, refer to that class for more details about the EDXML parsing interface.

Create a new EDXML parser. By default, the parser validates the input. Validation can be disabled by setting validate = False

Parameters **validate** (*bool, optional*) – Validate input or not

parse(*input_file*, *foreign_element_tags=()*)

Parses the specified file. The file can be any file-like object, or the name of a file that should be opened and parsed. The parser will generate calls to the various callback methods in the base class, allowing the parsed data to be processed.

Optionally, a list of tags of foreign elements can be supplied. The tags must prepend the namespace in James Clark notation. Example:

['{http://some/foreign/namespace}tag']

These elements will be passed to the _parse_foreign_element() when encountered.

Notes

Passing a file name rather than a file-like object is preferred and may result in a small performance gain.

Parameters

- **input_file** (*Union[io.TextIOBase, file, str]*) –
- **foreign_element_tags** (*List[str]*) –

Returns edxml.EDXMLPullParser

EDXMLOntologyPushParser

class edxml.**EDXMLOntologyPushParser**(*validate=True*, *foreign_element_tags=None*)
Bases: edxml.parser.EDXMLPushParser

A variant of the incremental push parser which ignores the events, parsing only the ontology information.

EDXMLOntologyPullParser

class edxml.**EDXMLOntologyPullParser**(*validate=True*)
Bases: edxml.parser.EDXMLPullParser

A variant of the incremental pull parser which ignores the events, parsing only the ontology information.

Create a new EDXML parser. By default, the parser validates the input. Validation can be disabled by setting validate = False

Parameters **validate** (*bool, optional*) – Validate input or not

EventCollection

class edxml.**EventCollection**(*events=()*, *ontology=None*)
Bases: list, typing.Generic

Class representing a collection of EDXML events. It is an extension of the list type and can be used like any other list.

Creates a new event collection, optionally initializing it with events and an ontology.

Parameters

- **events** (*Iterable[edxml.event.EDXMLEvent]*) – Initial event collection
- **ontology** (edxml.ontology.Ontology) – Corresponding ontology

extend(*iterable*)
Extend list by appending elements from the iterable.

create_dict_by_hash()
Creates a dictionary mapping sticky hashes to event collections containing the events that have that hash. The hashes are represented as hexadecimal strings.

Returns Dict[str, EventCollection]

is_equivalent_of(*other*)
Compares the collection with another specified collection. It returns True in case the two collections are equivalent, i.e. there are no semantic differences. For example, when one collection contains two instances of the same logical event while the other collection contains the result of merging the two events then there is no difference. Ordering of events or properties within an event are also irrelevant and do not result in any differences either.

Parameters **other** (EventCollection) – Another event collection

Returns bool

set_ontology(*ontology*)
Associates the evens in the collection to the specified EDXML ontology.

Parameters **ontology** (edxml.ontology.Ontology) –

Returns edxml.EventCollection

update_ontology(*ontology*)
Updates the ontology that is associated with the evens in the collection using the given ontology.

Parameters **ontology** (edxml.ontology.Ontology) –

resolve_collisions()
Returns a new EventCollection that contains only a single instance of each logical event in this collection. All input event instances that share a sticky hash are merged into a single output event.

Returns edxml.EventCollection

classmethod from_edxml(*edxml_data*, *foreign_element_tags=()*)
Parses EDXML data and returns a new EventSet containing the events and ontology information from the EDXML data.

Foreign elements are ignored by default. Optionally, tags of foreign elements can be specified allowing the parser to process them. The tags must prepend the namespace in James Clark notation. Example:

['{http://some/foreign/namespace}tag']

Parameters

- **edxml_data** (*bytes*) – The EDXML data
- **foreign_element_tags** (*Tuple[str]*) – Foreign element tags

Returns

Return type *EventCollection*

to_edxml(*pretty_print=True*)
Returns a string containing the EDXML representation of the events in the collection.

Parameters **pretty_print** (*bool*) – Pretty print output yes or no

Returns

Return type bytes

filter_type(*event_type_name*)
Returns a new event set containing the subset of events of specified event type.

Parameters **event_type_name** (*str*) –

Returns

Return type *EventCollection*

3.3 Filtering EDXML Data

A common task in EDXML event processing is filtering. When filtering data streams, the input is parsed, the parsed events and ontology are manipulated and re-serialized to the output. For this purpose the EDXML SDK features filtering classes. These classes are extensions of EDXML parsers that contain an EDXMLWriter instance to pass the parsed data through into the output. By subclassing one of the provided filtering classes, you can creep in between the parser and writer to alter the data in transit.

Using the filtering classes is best suited for tasks where the output ontology will be identical or highly similar to the input ontology. Some possible applications are:

- Deleting events from the input
- Deleting an event type (ontology and event data)
- Obfuscating sensitive data in input events

- Compressing the input by merging colliding events

Like with the parser classes, there is both a *push filter* and a *pull filter*, extending the push parser and pull parser respectively. In order to alter input data, the callback methods of the parser should be overridden. Then, the parent method can be called with a modified instance of the data that was passed to it.

3.3.1 Referencing Events

When storing references to parsed events you will notice that the events will each have their own EDXML namespace rather than inheriting from the root element. This is caused by the parser dereferencing the events after parsing, detaching them from the root element. To prevent that from happening the `copy()` method can be used to store a copy of the event in stead.

3.3.2 Class Documentation

The class documentation can be found below.

- *EDXMLFilterBase*
- *EDXMLPushFilter*
- *EDXMLPullFilter*

edxml.EDXMLFilterBase

class edxml.**EDXMLFilterBase**(*output*, *validate=True*)

Bases: edxml.parser.EDXMLParserBase

Extension of the EDXML parser that copies its input to the specified output. This class should not be instantiated. Instead, use one either EDXMLPullFilter or EDXMLPushFilter.

_writer = None

EDXML Writer

_close()

Callback that is invoked when the parsing process is finished or interrupted.

Returns The EDXML parser

Return type *EDXMLParserBase*

_parsed_ontology(*parsed_ontology*, *filtered_ontology=None*)

Callback that writes the parsed ontology into the output. By overriding this method and calling the parent method while passing a modified copy of the parsed ontology the output stream can be modified.

Parameters

- **parsed_ontology** (edxml.ontology.Ontology) – The input ontology
- **filtered_ontology** (edxml.ontology.Ontology) – The output ontology

_parsed_event(*event*)

Callback that writes the parsed event into the output. By overriding this method and calling the parent method after changing the event, the events in the output stream can be modified. If the parent method is not called, the event will be omitted in the output.

Parameters **event** (edxml.ParsedEvent) – The event

edxml.EDXMLPushFilter

class edxml.**EDXMLPushFilter**(*output*, *validate=True*)
Bases: edxml.parser.EDXMLPushParser, edxml.filter.EDXMLFilterBase

Extension of the push parser that copies its input to the specified output. By overriding the various callbacks provided by this class (or rather, the EDXMLFilterBase class), the EDXML data can be manipulated before the data is output.

edxml.EDXMLPullFilter

class edxml.**EDXMLPullFilter**(*output*, *validate=True*)
Bases: edxml.parser.EDXMLPullParser, edxml.filter.EDXMLFilterBase

Extension of the pull parser that copies its input to the specified output. By overriding the various callbacks provided by this class (or rather, the EDXMLFilterBase class), the EDXML data can be manipulated before the data is output.

3.4 Data Transcoding

EDXML data is most commonly generated from some type of input data. The input data is used to generate output events. The EDXML SDK features the concept of a *record transcoder*, which is a class that contains all required information and logic for transcoding a chunk of input data into an output event. The SDK can use record transcoders to generate events and event type definitions for you. It also facilitates unit testing of record transcoders.

In case the input data transforms into events of more than one event type, the transcoding process can be done my multiple record transcoders. This allows splitting the problem of transcoding the input in multiple parts. A *transcoder mediator* can be used to automatically route chunks of input data to the correct transcoder.

A record transcoder is an extension of the *EventTypeFactory* class. Because of this, record transcoders use class constants to describe event types. These class constants will be used to populate the output ontology while transcoding.

All record transcoders feature a *TYPE_MAP* constant which maps record selectors to event types. Record selectors identify chunks of input data that should transcode into a specific type of output event. What these selectors look like depends on the type of input data.

3.4.1 Object Transcoding

The most broadly usable record transcoder is the *ObjectTranscoder* class. Have a look at a quick example:

```
from edxml.transcode.object import ObjectTranscoder
from edxml_bricks.computing.generic import ComputingBrick


class UserTranscoder(ObjectTranscoder):
    TYPES = ['com.acme.staff.account']
    TYPE_MAP = {'user': 'com.acme.staff.account'}
    PROPERTY_MAP = {
        'com.acme.staff.account': {
            'name': 'user.name'
        }
    }
    TYPE_PROPERTIES = {
        'com.acme.staff.account': {
```

(continues on next page)

(continued from previous page)

```
            'user.name': ComputingBrick.OBJECT_USER_NAME
        }
    }
```

This minimal example shows a basic transcoder for input data records representing a user account. It shows the general structure of a transcoder, how to map input record types to output event types and map input record fields to output event properties.

The record selector shown in the *TYPE_MAP* constant is simply the name of type of input record: `user`. We will show how to label input records and how input record types are used to route input records to record transcoders later, when we discuss transcoder mediators.

The *PROPERTY_MAP* constant maps input record fields to event type properties. Just one field-to-property mapping is shown here, extending the example is straight forward.

The *TYPE_PROPERTIES* constant specifies the properties for each event type as well as the object type of each property. The object type that we refer to here using the `ComputingBrick.OBJECT_USER_NAME` constant is defined using an *ontology brick* from the public collection of ontology bricks.

Transcoding Steps

What happens when the *generate()* method of the transcoder is called is the following:

1. It will first generate an *ontology* using the *TYPES* constant to determine which event types to define and using the *PROPERTY_MAP* constant to look up which properties each event type should have. From the *TYPE_PROPERTIES* constant it determines the object types for each event property. In this example we have just one output event type which has a single property.
2. The transcoder will check the *TYPE_MAP* constant and see that an input record of type `user` should yield an output event of type `com.acme.staff.account`.
3. It checks the *PROPERTY_MAP* constant to see which record fields it should read and which property its values should be stored in. In this example, the `name` field goes into the `user.name` property.
4. The transcoder reads the `name` field from the input record and uses it to populate the `user.name` property.

Input Record Fields

We referred to `name` as a "field" because we did not specify what `name` refers to. Each input record might be a Python object and a field might actually be an attribute of the input record. Or the record might be a dictionary and the field a key in that dictionary. Both scenarios are supported. The transcoder will first try to treat the record as a dictionary and use its `get()` method to read the `name` item. In our example, the record is not a dictionary and the read will fail. Then, the transcoder will try to see if the record has an attribute named `name` by attempting to read it using the `getattr()` method. This will succeed and the output event property is populated.

Object Types and Concepts

Record transcoders usually define only event types, not the object types and concepts that these event types refer to. The reason is that these ontology elements are rarely specific for one particular transcoder. Object types and concepts are typically used my multiple transcoders and multiple data sources. In fact, object types and concepts are the very thing that enable machines to correlate information from multiple EDXML documents and forge it into a single consistent data set.

For that reason object types and concepts are usually defined by means of *ontology bricks* rather than a transcoder.

Selector Syntax

As we mentioned before, the `name` value in the *`PROPERTY_MAP`* constant is not just a name. It is a selector. As such, it can point to more than just dictionary entries or attributes in input records. Selectors support a dotted syntax to address values within values. For example, `foo.bar` can be used to access an item named `bar` inside a dictionary named `foo`. And if `bar` happens to be a list, you can address the first entry in that list by using `foo.bar.0` as selector.

Using a Mediator

Now we will extend the example to include a transcoder mediator:

```
import sys

from edxml.transcode.object import ObjectTranscoder, ObjectTranscoderMediator
from edxml_bricks.computing.generic import ComputingBrick


class UserTranscoder(ObjectTranscoder):
    TYPES = ['com.acme.staff.account']
    TYPE_MAP = {'user': 'com.acme.staff.account'}
    PROPERTY_MAP = {
        'com.acme.staff.account': {
            'name': 'user.name'
        }
    }
    TYPE_PROPERTIES = {
        'com.acme.staff.account': {
            'user.name': ComputingBrick.OBJECT_USER_NAME
        }
    }


class MyMediator(ObjectTranscoderMediator):
    TYPE_FIELD = 'type'


class Record:
    type = 'user'
    name = 'Alice'


with MyMediator(output=sys.stdout.buffer) as mediator:
    # Register the transcoder
    mediator.register('user', UserTranscoder())
    # Define an EDXML event source
    mediator.add_event_source('/acme/offices/amsterdam/')
    # Set the source as current source for all output events
    mediator.set_event_source('/acme/offices/amsterdam/')
    # Process the input record
    mediator.process(Record)
```

Now we see that the *`TYPE_FIELD`* constant of the mediator is used to set the name of the field in the input records that contains the record type. The example uses a single type of input record named *user*. The record transcoder is registered with the mediator using the same record type name. When the record is fed to the mediator using the *`process()`* method the mediator will read the record type and use the associated transcoder to generate an output

event. In this case the output EDXML stream will be written to standard output. Note that the transcoder produces binary data, so we write the output to `sys.stdout.buffer` rather than `sys.stdout`.

3.4.2 XML Transcoding

When you need to transcode XML input you can use the *XmlTranscoder* class. It is highly similar to the *ObjectTranscoder* class. The main difference is in how input records and fields are identified. This is done using XPath expressions. Below example illustrates this:

```
import sys
from io import BytesIO

from edxml.transcode.xml import XmlTranscoder, XmlTranscoderMediator
from edxml_bricks.computing.generic import ComputingBrick

# Define an input document
xml = bytes(
    '<records>'
    '  <users>'
    '    <user>'
    '      <name>Alice</name>'
    '    </user>'
    '  </users>'
    '</records>', encoding='utf-8'
)


# Define a transcoder for user records
class UserTranscoder(XmlTranscoder):
    TYPES = ['com.acme.staff.account']
    TYPE_MAP = {'.': 'com.acme.staff.account'}
    PROPERTY_MAP = {
        'com.acme.staff.account': {
            'name': 'user.name'
        }
    }
    TYPE_PROPERTIES = {
        'com.acme.staff.account': {
            'user.name': ComputingBrick.OBJECT_USER_NAME
        }
    }


# Transcode the input document
with XmlTranscoderMediator(output=sys.stdout.buffer) as mediator:
    # Register transcoder
    mediator.register('/records/users/user', UserTranscoder())
    # Define an EDXML event source
    mediator.add_event_source('/acme/offices/amsterdam/')
    # Set the source as current source for all output events
    mediator.set_event_source('/acme/offices/amsterdam/')
    # Parse the XML data
    mediator.parse(BytesIO(xml))
```

XPath expressions are used in various places in the above example. Firstly, the transcoder is registered to transcode all XML elements that are found using XPath expression `/records/users/user`. These will be treated as input records for the transcoding process. Note that transcoders should be registered using absolute XPath expressions.

Second, the *TYPE_MAP* constant indicates that all of the matching elements should be used for outputting an event of type `com.acme.staff.account`. This is achieved by using an XPath expression relative to the root of the input element. In our example the entire input element becomes the output event, so we use `.` as the XPath expression. In case the transcoder would produce various types of output events from sub-elements, then the XPath expressions in *TYPE_MAP* would need to select various sub-elements. Finally, the *PROPERTY_MAP* constant indicates that the event property named `user.name` should be populated by applying XPath expression `name` to the input record. This ultimately results in a single output event containing object value `Alice`.

Efficiency Considerations

The XML input is parsed in an incremental fashion. An in memory tree is built while parsing. For reasons of efficiency, the mediator will delete XML elements after transcoding them, thus preventing the in memory XML tree from growing while parsing. However, XML elements that have no matching transcoder are *not* deleted. The reason is that XML transcoders may need to aggregate information from multiple XML elements, while being registered on just one of those elements. So, the in memory XML tree may still grow when parts of the document have no transcoder associated with them.

The solution is to use the *NullTranscoder*. Registering this transcoder with an absolute XPath selector will allow the mediator to delete the matching XML elements and keep the in memory tree nice and clean.

3.4.3 Advanced Subjects

Defining Event Sources

The transcoder mediator examples showed how to add and select an event source for the output events. This will suffice for cases where the transcoder presents its output as a single EDXML event source. Transcoders may also define and use multiple EDXML event sources. This can be done by defining multiple sources and switch sources while feeding input records. For advanced use cases there are other methods that may suit you better.

Defining event sources can also be done by overriding the *generate_ontology()* method.

Assigning an event source to individual output events can be done by overriding the *post_process()* method. This is described in more detail *here*.

Outputting Multiple Events

By default, the transcoding process produces a single EDXML output event for each input data record. When input records contain a lot of information it may make sense to transcode a single input record into multiple output events. This can be achieved by overriding the *post_process()* function. This is explained in more detail in the next subsection.

Customizing Output Events

The events that are generated by the transcoder may be all you need for simple use cases. Sometimes the events may require additional processing. For example, you might need to adjust the values for some property. As an example, let us assume that the object values for some property need to be converted to lowercase. This can be done by defining the following generator:

```
def lowercase_user_name(name):
    yield name.lower()
```

Then, this generator can be used in the *TYPE_PROPERTY_POST_PROCESSORS* constant:

```
TYPE_PROPERTY_POST_PROCESSORS = {
    'com.acme.staff.account': {
        'user': lowercase_user_name,
    }
}
```

Note that the postprocessor can also be used to generate multiple object values from a single input record value.

If you need full access to the generated events and adjust them to your liking, then you can override the *post_process()* function. This function is actually a generator taking a single EDXML event as input and generating zero or more output events. The input record from which the event was constructed is provided as a function argument as well.

As an example, you might want to add the original JSON input record as an event attachment. An implementation of this could look like the following:

```
def post_process(self, event, input_record):
    event.attachments['input-record']['input-record'] = json.dumps(input_record)
    yield event
```

Customizing the Ontology

Not every aspect of the output ontology can be specified by means of the record transcoder class constants. Defining property relations is an example of this. Property relations are much better expressed in a procedural fashion. This can be done by overriding the *create_event_type()* class method. This is demonstrated in the following example:

```
@classmethod
def create_event_type(cls, event_type_name, ontology):

    user = super().create_event_type(event_type_name, ontology)

    user['name'].relate_intra('can be reached at', 'phone').because(
        "the LDAP directory entry of [[name]] mentions [[phone]]"
    )

    return user
```

Unit Testing

Record transcoders can be tested using a transcoder test harness. This is a special kind of transcoder mediator. There is the *TranscoderTestHarness* base class and the *ObjectTranscoderTestHarness* and *XmlTranscoderTestHarness* extensions. Feeding input records into these mediators will have the test harness use your transcoder to generate an EDXML document containing the output ontology and events, validating the output in the process. The EDXML document will then be parsed back into Python objects. The data is validated again in the process. Finally, any colliding events will be merged and the final event collection will be validated a third time. This assures that the output of the transcoder can be serialized into EDXML, deserialized and merged correctly.

After feeding the input records the parsed ontology and events are exposed by means of the *events* attribute of the test harness. This attribute is an *EventCollection* which you can use to write assertions about the resulting ontology and events. So, provided you feed the test harness with a good set of test records, this results in unit tests that cover everything. The transcoding process itself, ontology generation, validity of the output events and event merging logic.

A full example of the use of a test harness is shown below:

```
import pytest

from edxml.transcode.object import ObjectTranscoderTestHarness, ObjectTranscoder
from edxml_bricks.computing.generic import ComputingBrick


class TestObjectTranscoder(ObjectTranscoder):
    __test__ = False

    TYPES = ['com.acme.staff.account']
    TYPE_MAP = {'user': 'com.acme.staff.account'}
    PROPERTY_MAP = {
        'com.acme.staff.account': {
            'name': 'user.name'
        }
    }
    TYPE_PROPERTIES = {
        'com.acme.staff.account': {
            'user.name': ComputingBrick.OBJECT_USER_NAME
        }
    }


@pytest.fixture()
def fixture_object():
    return {'type': 'user', 'name': 'Alice'}


def test(fixture_object):
    with ObjectTranscoderTestHarness(TestObjectTranscoder(), record_selector='type')␣
↪as harness:
        harness.process_object(fixture_object)

    assert harness.events[0]['user.name'] == {'Alice'}
```

Automatic Data Normalization and Cleaning

By default a transcoder will just copy values from the input records into the properties of the output events. Often times this will not yield the desired result. A common case is date / time values. There are many different formats for representing time, and EDXML accepts just one specific format. Even greater challenges arise when the types of values contained in a single input record field may vary from one input record to another. Or when an input record field may occasionally contain downright nonsensical gibberish.

Transcoders feature various means of dealing with these challenges. Input data can be automatically normalized and cleaned. The default transcoder behavior is to error when it produces an event containing an invalid object value. In stead of adding code to properly normalize event object values in your transcoders, you can also have the SDK do the normalization for you. In order to do so you can use the *TYPE_AUTO_REPAIR_NORMALIZE* constant to opt into automatic normalization for a particular event property.

Automatic normalization also supports events containing non-string values. For example, placing a float in a property that uses an EDXML data type from the decimal family can automatically normalize that float into the proper decimal string representation that fits the data type. Some of the supported Python types are float, bool, datetime, Decimal and IP (from IPy).

In some cases, input data may contain values that cannot be normalized automatically. Using the *TYPE_AUTO_REPAIR_DROP* constant it is possible to opt into dropping these values from the output event. A

common case is an input record that contains a field that may hold both IPv4 and IPv6 internet addresses. In EDXML these must be represented using different data types and separate event properties. This can be done by having the transcoder store the value in both properties. This means that one of the two properties will always contain an invalid value. By allowing these invalid values to be dropped, the output events will always have the values in the correct event property.

Note that there can be quite a performance penalty for enabling automatic normalization and cleaning. In many cases, this will not matter much. The transcoder is optimistic. As long as the output events are valid no normalization or cleaning done and there is no performance hit. Only when an output event fails to validate the expensive event repair code is run.

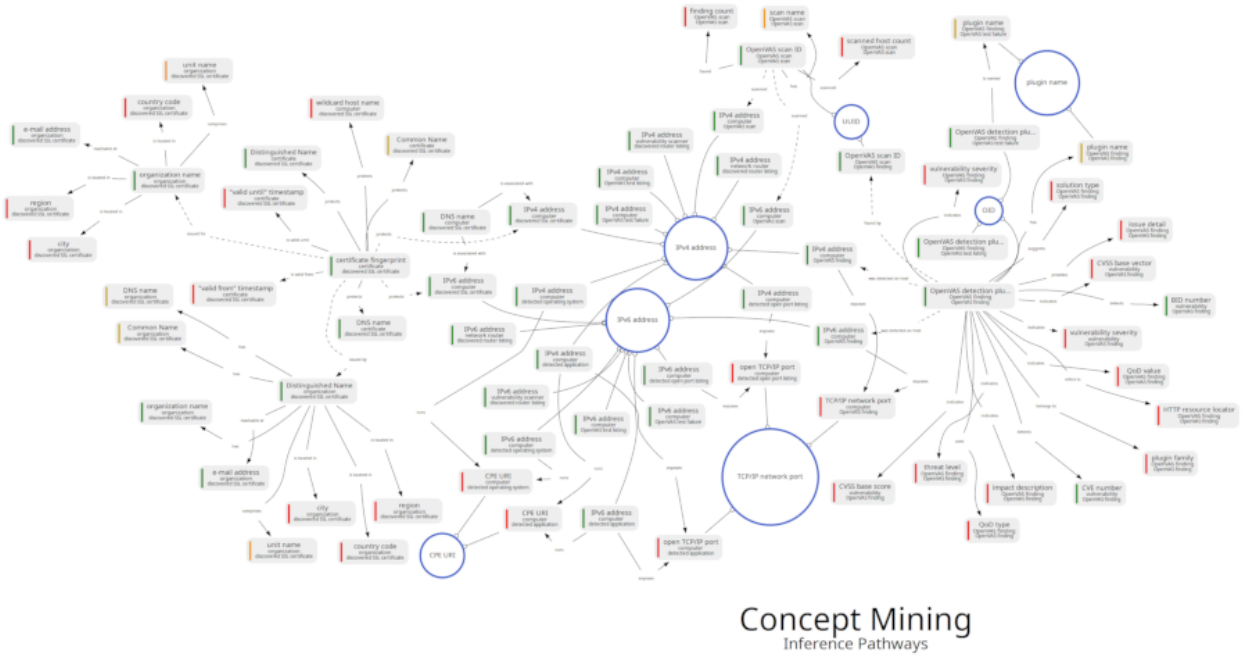
In case performance turns out to be an issue, you can always optimize your transcoder by normalizing event objects yourself. You might find the `TYPE_PROPERTY_POST_PROCESSORS` constant helpful. Alternatively, you can override the `post_process()` function to modify the autogenerated events as needed.

In case you want to retain the original record values as they were before normalization and cleaning there are two options for doing so. Firstly, the original value could be stored in another property and an 'original' relations could be used to relate the original value to the normalized one. Second, (part of) the original input record could be stored as an event attachment.

Description & Visualization

The *Transcoder Mediator* class contains two methods that allows EDXML transcoders to generate descriptions and visualizations of their output ontology. Both can be great aids to review the ontology information in your record transcoders. The first of these methods is `describe_transcoder()`. Refer to that method for details. The other method is `generate_graphviz_concept_relations()`. Again, refer to that method for details.

The image displayed below shows an example of the output of `generate_graphviz_concept_relations()` for an advanced transcoder that employs multiple record transcoders:



The image shows the various reasoning pathways for *concept mining* provided by the full ontology of all record transcoders combined. It tells the transcoder developer how machines can correlate information to mine knowledge from the data.

3.4.4 API Documentation

Below, the documentation of the various transcoding classes is given.

Base Classes

- *Record Transcoder*
- *Transcoder Mediator*
- *Transcoder Test Harness*

Transcoders & Mediators

- *Object Transcoder*
- *Object Transcoder Mediator*
- *XML Transcoder*
- *XML Transcoder Mediator*

Test Harnesses

- *Object Transcoder Test Harness*
- *XML Transcoder Test Harness*

RecordTranscoder

class edxml.transcode.**RecordTranscoder**

Bases: edxml.ontology.event_type_factory.EventTypeFactory

This is a base class that can be extended to implement record transcoders for the various input data record types that are processed by a particular TranscoderMediator implementation. The class extends the EventTypeFactory class, which is used to generate the event types for the events that will be produced by the record transcoder.

TYPE_MAP = {}

The TYPE_MAP attribute is a dictionary mapping input record type selectors to the corresponding EDXML event type names. This mapping is used by the transcoding mediator to find the correct record transcoder for each input data record.

Note: When no EDXML event type name is specified for a particular input record type selector, it is up to the record transcoder to set the event type on the events that it generates.

Note: The fallback record transcoder must set the None key to the name of the EDXML fallback event type.

PROPERTY_MAP = {}

The PROPERTY_MAP attribute is a dictionary mapping event type names to the property value selectors for finding property objects in input records. Each value in the dictionary is another dictionary that maps value selectors to property names. The exact nature of the value selectors differs between record transcoder implementations.

TYPE_PROPERTY_POST_PROCESSORS = {}

The TYPE_PROPERTY_POST_PROCESSORS attribute is a dictionary mapping EDXML event type names to property processors. The property processors are a dictionary mapping property names to processors. A processor is a function that accepts a value from the input field that corresponds with the

property and returns an iterable yielding zero or more values which will be stored in the output event. The processors will be applied to input record values before using them to create output events.

Example:

```
{
  'event-type-name': {
    'property-a': lambda x: yield x.lower()
  }
}
```

TYPE_AUTO_REPAIR_NORMALIZE = {}

The TYPE_AUTO_REPAIR_NORMALIZE attribute is a dictionary mapping EDXML event type names to properties which should be repaired automatically by normalizing their object values. This means that the transcoder is not required to store valid EDXML string representations in its output events. Rather, it may store any type of value which can be normalized into valid string representations automatically. Please refer to the *normalize_objects()* method for a list of supported value types. The names of properties for which values may be normalized are specified as a list. Example:

```
{'event-type-name': ['some-property']}
```

TYPE_AUTO_REPAIR_DROP = {}

The TYPE_AUTO_REPAIR_DROP attribute is a dictionary mapping EDXML event type names to properties which should be repaired automatically by dropping invalid object values. This means that the transcoder is permitted to store object values which cause the output event to be invalid. The EDXML writer will attempt to repair invalid output events. First, it will try to normalize object values when configured to do so. As a last resort, it can try to drop any offending object values. The names of properties for which values may be dropped are specified as a list. Example:

```
{'event-type-name': ['some-property']}
```

generate(*record*, *record_selector*, ***kwargs*)

Generates one or more EDXML events from the given input record

Parameters

- **record** – Input data record
- **record_selector** (*str*) – The selector matching the input record
- ****kwargs** – Arbitrary keyword arguments

Yields *edxml.EDXMLEvent*

post_process(*event*, *input_record*)

Generates zero or more EDXML output events from the given EDXML input event. If this method is overridden by an extension of the RecordTranscoder class, all events generated by the generate() method are passed through this method for post processing. This allows the generated events to be modified or omitted. Or, multiple derivative events can be created from a single input event.

The input record that was used to generate the input event is also passed as a parameter. Post processors can use this to extract additional information and add it to the input event.

Parameters

- **event** (edxml.EDXMLEvent) – Input event
- **input_record** – Input record

Yields *edxml.EDXMLEvent*

TranscoderMediator

class edxml.transcode.**TranscoderMediator**(*output=None*)
Bases: object

Base class for implementing mediators between a non-EDXML input data source and a set of RecordTranscoder implementations that can transcode the input data records into EDXML events.

Sources can instantiate the mediator and feed it input data records, while record transcoders can register themselves with the mediator in order to transcode the types of input record that they support.

The class is a Python context manager which will automatically flush the output buffer when the mediator goes out of scope.

Create a new transcoder mediator which will output streaming EDXML data using specified output. The output parameter is a file-like object that will be used to send the XML data to. Note that the XML data is binary data, not string data. When the output parameter is omitted, the generated XML data will be returned by the methods that generate output.

Parameters **output** (*file, optional*) – a file-like object

register(*record_selector*, *record_transcoder*)
Register a record transcoder for processing records identified by the specified record selector. The exact nature of the record selector depends on the mediator implementation.

The same record transcoder can be registered for multiple record selectors.

Note: Any record transcoder that registers itself as a transcoder using None as selector is used as the fallback record transcoder. The fallback record transcoder is used to transcode any record for which no transcoder has been registered.

Parameters

- **record_selector** – Record type selector
- **record_transcoder** (edxml.transcode.RecordTranscoder) – Record transcoder instance

debug(*warn_no_transcoder=True*, *warn_fallback=True*, *log_repaired_events=True*)
Enable debugging mode, which prints informative messages about transcoding issues, disables event buffering and stops on errors.

Using the keyword arguments, specific debug features can be disabled. When warn_no_transcoder is set to False, no warnings will be generated when no matching record transcoder can be found. When warn_fallback is set to False, no warnings will be generated when an input record is routed to the fallback transcoder. When log_repaired_events is set to False, no message will be generated when an invalid event was repaired.

Parameters

- **warn_no_transcoder** (*bool*) – Warn when no record transcoder found
- **warn_fallback** (*bool*) – Warn when using fallback transcoder
- **log_repaired_events** (*bool*) – Log events that were repaired

Returns

Return type *TranscoderMediator*

disable_event_validation()
Instructs the EDXML writer not to validate its output. This may be used to boost performance in case you know that the data will be validated at the receiving end, or in case you know that your generator is perfect. :)

Returns

Return type *TranscoderMediator*

enable_auto_repair_drop(*event_type_name*, *property_names*)
Allows dropping invalid object values from the specified event properties while repairing invalid events. This will only be done as a last resort when normalizing object values failed or is disabled.

Note: Dropping object values may still lead to invalid events.

Parameters

- **event_type_name** (*str*) –
- **property_names** (*List[str]*) –

Returns

Return type *TranscoderMediator*

ignore_invalid_events(*warn=False*)
Instructs the EDXML writer to ignore invalid events. After calling this method, any event that fails to validate will be dropped. If warn is set to True, a detailed warning will be printed, allowing the source and cause of the problem to be determined.

Note: If automatic event repair is enabled the writer will attempt to repair any invalid events before dropping them.

Note: This has no effect when event validation is disabled.

Parameters **warn** (*bool*) – Log warnings or not

Returns

Return type *TranscoderMediator*

ignore_post_processing_exceptions(*warn=True*)
Instructs the mediator to ignore exceptions raised by the _post_process() methods of record transcoders. After calling this method, any input record that that fails transcode due to post processing errors will be ignored and a warning is logged. If warn is set to False these warnings are suppressed.

Parameters **warn** (*bool*) – Log warnings or not

Returns

Return type *TranscoderMediator*

enable_auto_repair_normalize(*event_type_name*, *property_names*)
Enables automatic repair of the property values of events of specified type. Whenever an invalid event is generated by the mediator it will try to repair the event by normalizing object values of specified properties.

Parameters

- **event_type_name** (*str*) –
- **property_names** (*List[str]*) –

Returns TranscoderMediator

add_event_source(*source_uri*)

Adds an EDXML event source definition. If no event sources are added, a bogus source will be generated.

> **Warning:** The source URI is used to compute sticky hashes. Therefore, adjusting the source URIs of events after generating them changes their hashes.

The mediator will not output the EDXML ontology until it receives its first event through the process() method. This means that the caller can generate an event source 'just in time' by inspecting the input record and use this method to create the appropriate source definition.

Returns the created EventSource instance, to allow it to be customized.

Parameters **source_uri** (*str*) – An Event Source URI

Returns

Return type *EventSource*

set_event_source(*source_uri*)

Set a fixed event source for the output events. This source will automatically be set on every output event.

Parameters **source_uri** (*str*) – The event source URI

Returns

Return type *TranscoderMediator*

generate_graphviz_concept_relations()

Returns a graph that shows possible concept mining reasoning paths.

Returns graphviz.Digraph

describe_transcoder(*input_description*)

Returns a reStructuredText description for a transcoder that uses this mediator. This is done by combining the ontologies of all registered record transcoders and describing what the resulting data would entail.

Parameters **input_description** (*str*) – Short description of the input data

Returns str

process(*record*)

Processes a single input record, invoking the correct transcoder to generate an EDXML event and writing the event into the output.

If no output was specified while instantiating this class, any generated XML data will be returned as bytes.

Parameters **record** – Input data record

Returns Generated output XML data

Return type bytes

close(*write_ontology_update=True*)

Finalizes the transcoding process by flushing the output buffer. When the mediator is not used as a context manager, this method must be called explicitly to properly close the mediator.

By default the current ontology will be written to the output if needed. This can be prevented by using the method parameter.

If no output was specified while instantiating this class, any generated XML data will be returned as bytes.

Parameters **write_ontology_update** (*bool*) – Output ontology yes/no

Returns Generated output XML data

Return type bytes

TranscoderTestHarness

class edxml.transcode.**TranscoderTestHarness**(*transcoder*, *record_selector*, *base_ontology=None*, *register=True*)

Bases: edxml.transcode.mediator.TranscoderMediator

This class is a substitute for the transcoding mediators which can be used to test record transcoders. It provides the means to feed input records to record transcoders and make assertions about the output events.

After processing is completed, either by closing the context or by explicitly calling close(), any colliding events are merged. This means that unit tests will also test the merging logic of the events.

Creates a new test harness for testing specified record transcoder. Optionally a base ontology can be provided. When provided, the harness will try to upgrade the base ontology to the ontology generated by the record transcoder, raising an exception in case of backward incompatibilities.

By default the record transcoder will be automatically registered at the specified selector. In case you wish to do the record registration on your own you must set the register parameter to False.

Parameters

- **transcoder** (edxml.transcode.RecordTranscoder) – The record transcoder under test
- **record_selector** (*str*) – Record type selector
- **base_ontology** (edxml.ontology.Ontology) – Base ontology
- **register** (*bool*) – Register the transcoder yes or no

events = None

The resulting event collection

process(*record*, *selector=None*)

Processes a single record, invoking the correct record transcoder to generate an EDXML event and adding the event to the event set.

The event is also written into an EDXML writer and parsed back to an event object. This means that all validation that would be applied to the event when using the real transcoder mediator has been applied.

Parameters

- **record** – The input record
- **selector** – The selector that matched the record

close(*write_ontology_update=True*)

Finalizes the transcoding process by flushing the output buffer. When the mediator is not used as a context manager, this method must be called explicitly to properly close the mediator.

By default the current ontology will be written to the output if needed. This can be prevented by using the method parameter.

If no output was specified while instantiating this class, any generated XML data will be returned as bytes.

Parameters **write_ontology_update** (*bool*) – Output ontology yes/no

Returns Generated output XML data

Return type bytes

ObjectTranscoder

class edxml.transcode.object.**ObjectTranscoder**

Bases: edxml.transcode.transcoder.RecordTranscoder

PROPERTY_MAP = {}

The PROPERTY_MAP attribute is a dictionary mapping event type names to their associated property mappings. Each property mapping is itself a dictionary mapping input record attribute names to EDXML event properties. The map is used to automatically populate the properties of the output events produced by the generate() method of the ObjectTranscoder class. The attribute names may contain dots, indicating a subfield or positions within a list, like so:

```
{'event-type-name': {'fieldname.0.subfieldname': 'property-name'}}
```

Mapping field values to multiple event properties is also possible:

```
{'event-type-name': {'fieldname.0.subfieldname': ['property', 'another-
↪property']}}
```

Note that the event structure will not be validated until the event is yielded by the generate() method. This creates the possibility to add nonexistent properties to the attribute map and remove them in the generate() method, which may be convenient for composing properties from multiple input record attributes, or for splitting the auto-generated event into multiple output events.

EMPTY_VALUES = {}

The EMPTY_VALUES attribute is a dictionary mapping input record fields to values of the associated property that should be considered empty. As an example, the data source might use a specific string to indicate a value that is absent or irrelevant, like '-', 'n/a' or 'none'. By listing these values with the field associated with an output event property, the property will be automatically omitted from the generated EDXML events. Example:

```
{'fieldname.0.subfieldname': ('none', '-')}
```

Note that empty values are *always* omitted, because empty values are not permitted in EDXML event objects.

generate(*input_object*, *record_selector*, ***kwargs*)

Generates one or more EDXML events from the given input record, populating it with properties using the PROPERTY_MAP class property.

When the record transcoder is the fallback transcoder, record_selector will be None.

The input record can be a dictionary or act like one, it can be an object, a dictionary containing objects or an object containing dictionaries. Object attributes or dictionary items are allowed to contain lists or other objects. The keys in the PROPERTY_MAP will be used to access its items or attributes. Using dotted notation in PROPERTY_MAP, you can extract pretty much everything from anything.

This method can be overridden to create a generic event generator, populating the output events with generic properties that may or may not be useful to the specific record transcoders. The specific record transcoders can refine the events that are generated upstream by adding, changing or removing properties, editing the event attachments, and so on.

Parameters

- **input_object** (*dict, object*) – Input object
- **record_selector** (*Optional[str]*) – The name of the input record type
- ****kwargs** – Arbitrary keyword arguments

Yields *EDXMLEvent*

ObjectTranscoderMediator

class edxml.transcode.object.**ObjectTranscoderMediator**(*output=None*)

Bases: edxml.transcode.mediator.TranscoderMediator

This class is a mediator between a source of Python objects, also called input records, and a set of ObjectTranscoder implementations that can transcode the objects into EDXML events.

Sources can instantiate the mediator and feed it records, while record transcoders can register themselves with the mediator in order to transcode the record types that they support. Note that we talk about "record types" rather than "object types" because mediators distinguish between types of input record by inspecting the attributes of the object rather than inspecting the Python object as obtained by calling type() on the object.

Create a new transcoder mediator which will output streaming EDXML data using specified output. The output parameter is a file-like object that will be used to send the XML data to. Note that the XML data is binary data, not string data. When the output parameter is omitted, the generated XML data will be returned by the methods that generate output.

Parameters **output** (*file, optional*) – a file-like object

TYPE_FIELD = None

This constant must be set to the name of the item or attribute in the object that contains the input record type, allowing the TranscoderMediator to route objects to the correct record transcoder.

If the constant is set to None, all objects will be routed to the fallback transcoder. If there is no fallback transcoder available, the record will not be processed.

Note: The fallback transcoder is a record transcoder that registered itself using None as record type.

---

register(*record_type_identifier*, *transcoder*)

Register a record transcoder for processing objects of specified record type. The same record transcoder can be registered for multiple record types.

Note: Any record transcoder that registers itself using None as record_type_identifier is used as the fallback transcoder. The fallback transcoder is used to transcode any record for which no record transcoder has been registered.

Parameters

- **record_type_identifier** (*Optional[str]*) – Name of the record type
- **transcoder** (ObjectTranscoder) – ObjectTranscoder class

process(*input_record*)

Processes a single input object, invoking the correct object transcoder to generate an EDXML event and writing the event into the output.

If no output was specified while instantiating this class, any generated XML data will be returned as bytes.

The object may optionally be a dictionary or act like one. Object transcoders can extract EDXML event object values from both dictionary items and object attributes as listed in the PROPERTY_MAP of the matching record transcoder. Using dotted notation the keys in PROPERTY_MAP can refer to dictionary items or object attributes that are themselves dictionaries of lists.

Parameters **input_record** (*dict, object*) – Input object

Returns Generated output XML data

Return type bytes

XmlTranscoder

class edxml.transcode.xml.**XmlTranscoder**

Bases: edxml.transcode.transcoder.RecordTranscoder

TYPE_MAP = {}

The TYPE_MAP attribute is a dictionary mapping XPath expressions to EDXML event type names. The XPath expressions are relative to the XPath of the elements that that record transcoder is registered to at the transcoding mediator. The expressions in TYPE_MAP are evaluated on each XML input element to obtain sub-elements. For each sub-element an EDXML event of the corresponding type is generated. In case the events are supposed to be generated from the input element as a whole, you can use '.' for the XPath expression. However, you can also use the expressions to produce multiple types of output events from different parts of the input element.

Note: When no EDXML event type name is specified for a particular XPath expression, it is up to the record transcoder to set the event type on the events that it generates.

Note: The fallback transcoder must set the None key to the name of the EDXML fallback event type.

Example

{'.': 'some-event-type'}

PROPERTY_MAP = {}

The PROPERTY_MAP attribute is a dictionary mapping event type names to the XPath expressions for finding property objects. Each value in the dictionary is another dictionary that maps property names to the XPath expression. The XPath expressions are relative to the source XML element of the event. Example:

```
{'event-type-name': {'some/subtag[@attribute]': 'property-name'}}
```

The use of EXSLT regular expressions is supported and may be used in Xpath keys like this:

```
{'event-type-name': {'*[re:test(., "^abc$", "i")]': 'property-name'}}
```

Mapping XPath expressions to multiple event properties is also possible:

```
{'event-type-name': {'some/subtag[@attribute]': ['property', 'another-property
↪']}}
```

Extending XPath by injecting custom Python functions is supported due to the lxml implementation of XPath that is being used in the record transcoder implementation. Please refer to the lxml documentation about this subject. This record transcoder implementation provides a small set of custom XPath functions already, which shows how it is done.

Note that the event structure will not be validated until the event is yielded by the generate() method. This creates the possibility to add nonexistent properties to the XPath map and remove them in the Generate method, which may be convenient for composing properties from multiple XML input tags or attributes, or for splitting the auto-generated event into multiple output events.

EMPTY_VALUES = {}

The EMPTY_VALUES attribute is a dictionary mapping XPath expressions to values of the associated property that should be considered empty. As an example, the data source might use a specific string to indicate a value that is absent or irrelevant, like '-', 'n/a' or 'none'. By listing these values with the XPath expression associated with an output event property, the property will be automatically omitted from the generated EDXML events. Example:

```
{'./some/subtag[@attribute]': ('none', '-')}
```

Note that empty values are *always* omitted, because empty values are not permitted in EDXML event objects.

generate(*element*, *record_selector*, ***kwargs*)

Generates one or more EDXML events from the given XML element, populating it with properties using the PROPERTY_MAP class property.

When the record transcoder is the fallback transcoder, record_selector will be None.

This method can be overridden to create a generic event generator, populating the output events with generic properties that may or may not be useful to the specific record transcoders. The specific record transcoders can refine the events that are generated upstream by adding, changing or removing properties, editing the event content, and so on.

Parameters

- **element** (*etree.Element*) – XML element
- **record_selector** (*Optional[str]*) – The matching XPath selector
- ****kwargs** – Arbitrary keyword arguments

Yields *EDXMLEvent*

XmlTranscoderMediator

class edxml.transcode.xml.**XmlTranscoderMediator**(*output=None*)

Bases: edxml.transcode.mediator.TranscoderMediator

This class is a mediator between a source of XML elements and a set of XmlTranscoder implementations that can transcode the XML elements into EDXML events.

Sources can instantiate the mediator and feed it XML elements, while record transcoders can register themselves with the mediator in order to transcode the types of XML element that they support.

register(*xpath_expression*, *transcoder*, *tag=None*)

Register a record transcoder for processing XML elements matching specified XPath expression. The same record transcoder can be registered for multiple XPath expressions. The transcoder argument must be a XmlTranscoder class or an extension of it.

The optional tag argument can be used to pass a list of tag names. Only the tags in the input XML data that are included in this list will be visited while parsing and matched against the XPath expressions associated

with registered record transcoders. When the argument is not used, the tag names will be guessed from the xpath expressions that the record transcoders have been registered with. Namespaced tags can be specified using James Clark notation:

```
{http://www.w3.org/1999/xhtml}html
```

The use of EXSLT regular expressions in XPath expressions is supported and can be specified like in this example:

```
*[re:test(., "^abc$", "i")]
```

Note: Any record transcoder that registers itself using None as the XPath expression is used as the fallback transcoder. The fallback transcoder is used to transcode any record that does not match any XPath expression of any registered transcoder.

Parameters

- **xpath_expression** (*Optional[str]*) – XPath of matching XML records
- **transcoder** (XmlTranscoder) – XmlTranscoder
- **tag** (*Optional[str]*) – XML tag name

parse(*input_file*, *attribute_defaults=False*, *dtd_validation=False*, *load_dtd=False*, *no_network=True*, *remove_blank_text=False*, *remove_comments=False*, *remove_pis=False*, *encoding=None*, *html=False*, *recover=None*, *huge_tree=False*, *schema=None*, *resolve_entities=False*)

Parses the specified file, writing the resulting EDXML data into the output. The file can be any file-like object, or the name of a file that should be opened and parsed.

The other keyword arguments are passed directly to `lxml.etree.iterparse`, please refer to the lxml documentation for details.

If no output was specified while instantiating this class, any generated XML data will be collected in a memory buffer and returned when the transcoder is closed.

Notes

Passing a file name rather than a file-like object is preferred and may result in a small performance gain.

Parameters

- **schema** – an XMLSchema to validate against
- **huge_tree** (*bool*) – disable security restrictions and support very deep trees and very long text content (only affects libxml2 2.7+)
- **recover** (*bool*) – try hard to parse through broken input (default: True for HTML, False otherwise)
- **html** (*bool*) – parse input as HTML (default: XML)
- **encoding** – override the document encoding
- **remove_pis** (*bool*) – discard processing instructions
- **remove_comments** (*bool*) – discard comments
- **remove_blank_text** (*bool*) – discard blank text nodes

- **no_network** (*bool*) – prevent network access for related files
- **load_dtd** (*bool*) – use DTD for parsing
- **dtd_validation** (*bool*) – validate (if DTD is available)
- **attribute_defaults** (*bool*) – read default attributes from DTD
- **resolve_entities** (*bool*) – replace entities by their text value (default: True)
- **input_file** (*Union[io.TextIOBase, file, str]*) –

generate(*input_file*, *attribute_defaults=False*, *dtd_validation=False*, *load_dtd=False*, *no_network=True*, *remove_blank_text=False*, *remove_comments=False*, *remove_pis=False*, *encoding=None*, *html=False*, *recover=None*, *huge_tree=False*, *schema=None*, *resolve_entities=False*)

Parses the specified file, yielding bytes containing the resulting EDXML data while parsing. The file can be any file-like object, or the name of a file that should be opened and parsed.

If an output was specified when instantiating this class, the EDXML data will be written into the output and this generator will yield empty strings.

The other keyword arguments are passed directly to `lxml.etree.iterparse`, please refer to the lxml documentation for details.

Notes

Passing a file name rather than a file-like object is preferred and may result in a small performance gain.

Parameters

- **schema** – an XMLSchema to validate against
- **huge_tree** (*bool*) – disable security restrictions and support very deep trees and very long text content (only affects libxml2 2.7+)
- **recover** (*bool*) – try hard to parse through broken input (default: True for HTML, False otherwise)
- **html** (*bool*) – parse input as HTML (default: XML)
- **encoding** – override the document encoding
- **remove_pis** (*bool*) – discard processing instructions
- **remove_comments** (*bool*) – discard comments
- **remove_blank_text** (*bool*) – discard blank text nodes
- **no_network** (*bool*) – prevent network access for related files
- **load_dtd** (*bool*) – use DTD for parsing
- **dtd_validation** (*bool*) – validate (if DTD is available)
- **attribute_defaults** (*bool*) – read default attributes from DTD
- **resolve_entities** (*bool*) – replace entities by their text value (default: True)
- **input_file** (*Union[io.TextIOBase, file, str]*) –

Yields *bytes* – Generated output XML data

process (*element*, *tree=None*)
Processes a single XML element, invoking the correct record transcoder to generate an EDXML event and writing the event into the output.

If no output was specified while instantiating this class, any generated XML data will be returned as bytes.

Parameters

- **element** (*etree.Element*) – XML element
- **tree** (*etree.ElementTree*) – Root of XML document being parsed

Returns Generated output XML data

Return type bytes

static get_visited_tag_name (*xpath*)
Tries to determine the name of the tag of elements that match the specified XPath expression. Raises ValueError in case the xpath expression is too complex to determine the tag name.

Returns Optional[List[str]]

ObjectTranscoderTestHarness

class edxml.transcode.object.**ObjectTranscoderTestHarness** (*transcoder*, *record_selector*, *base_ontology=None*, *register=True*)
Bases: edxml.transcode.test_harness.TranscoderTestHarness

Creates a new test harness for testing specified record transcoder. When provided, the harness will try to upgrade the base ontology to the ontology generated by the record transcoder, raising an exception in case of backward incompatibilities.

The record_selector is the selector that the record transcoder will be registered at.

By default the record transcoder will be automatically registered at the specified xpath. In case you wish to do the record registration on your own you must set the register parameter to False.

Parameters

- **transcoder** (edxml.transcode.RecordTranscoder) – The record transcoder under test
- **base_ontology** (*edxml.Ontology*) – Base ontology
- **register** (*bool*) – Register the transcoder yes or no

process_object (*fixture_object*, *selector=None*, *close=True*)
Parses specified object and transcodes it to produce output events. The output events are added to the event set.

The selector is only relevant when the record transcoder can output multiple types of events. It must be set to the selector of the sub-object inside the object being transcoded that corresponds with one specific output event type. When unspecified and the transcoder produces a single type of output events it will be fetched from the TYPE_MAP constant of the record transcoder.

By default, the test harness is automatically closed after processing the record. When processing multiple input records this can be prevented. Note that the harness must be closed before using it in assertions.

Parameters

- **fixture_object** – The object to use as input record fixture

- **selector** –
- **close** (*bool*) – Close test harness after processing yes / no

XmlTranscoderTestHarness

class edxml.transcode.xml.**XmlTranscoderTestHarness**(*fixtures_path*, *transcoder*, *transcoder_root*, *base_ontology=None*, *register=True*)

Bases: edxml.transcode.test_harness.TranscoderTestHarness

Creates a new test harness for testing specified record transcoder using XML fixtures stored at the indicated path. When provided, the harness will try to upgrade the base ontology to the ontology generated by the record transcoder, raising an exception in case of backward incompatibilities.

The transcoder_root is the XPath expression that the record transcoder will be registered at. It will be used to extract the input elements for the record transcoder from XML fixtures, exactly as XmlTranscoderMediator would do on real data.

By default the record transcoder will be automatically registered at the specified xpath. In case you wish to do the record registration on your own you must set the register parameter to False.

Parameters

- **fixtures_path** (*str*) – Path to the fixtures set
- **transcoder** (edxml.transcode.RecordTranscoder) – The record transcoder under test
- **transcoder_root** (*str*) – XPath for record transcoder registration
- **base_ontology** (*edxml.Ontology*) – Base ontology
- **register** (*bool*) – Register the transcoder yes or no

process_xml(*filename*, *element_root=None*)

Parses specified XML file and transcodes it to produce output events. The output events are added to the event set. The filename argument must be a path relative to the fixtures path.

The XML file is expected to be structured like real input data stripped down to contain the XML elements that are relevant to the record transcoder under test.

The element_root is only relevant when the record transcoder can output multiple types of events. It must be set to the XPath expression of the sub-element inside the element being transcoded that corresponds with one specific output event type. When unspecified and the transcoder produces a single type of output events it will be fetched from the TYPE_MAP constant of the record transcoder.

Parameters

- **filename** (*str*) – The XML file to use as input record fixture
- **element_root** –

NullTranscoder

class edxml.transcode.**NullTranscoder**

Bases: edxml.transcode.transcoder.RecordTranscoder

This is a pseudo-transcoder that is used to indicate input records that should be discarded rather than transcoder into output events. By registering this transcoder for transcoding particular types of input records, those records will be ignored.

3.5 EDXML Event Representations

EDXML events are represented as instances of the *EDXMLEvent* class and its subclasses.

3.5.1 Accessing Properties

The event properties are accessible by means of the `properties` attribute. The objects of each property are stored as sets. To make accessing properties as convenient as possible, events implement a MutableMapping, allowing more direct access. A quick example:

```
from edxml import EDXMLEvent

# Create an event
event = EDXMLEvent(properties={'names': {'Alice', 'Bob'}})

for property_name, object_values in event.items():
    for object_value in object_values:
        print(object_value)
```

Note that the event that we created above is incomplete. It is not even valid. We did not set an event type or an event source. EDXMLEvent instances are not bound to an ontology. As such there is no concept of validity in its instances. You can create any event you like, valid or invalid. Validating an instance can be done by means of its `is_valid()` method, which accepts an ontology as parameter.

Due to events not being bound to an ontology there is no differentiation between a property that does not exist and a property that exists but is lacking any objects. Therefore, checking if the properties dictionary has a certain key will return `False` even if that key has been assigned an empty set.

Writing event properties works just as you would expect. Some examples are shown below.

```
from edxml import EDXMLEvent

# Create an event
event = EDXMLEvent()

# Assign properties in one go
event.properties = {'names': {'Alice', 'Bob'}}

# Single values are wrapped into a set automatically
event.properties['names'] = 'Alice'

# The above could be shortened like this:
event['names'] = 'Alice'

# Add an object value
event['names'].add('Bob')

# Clear property
del event['names']
```

3.5.2 Accessing Attachments

Event attachments can be accessed by means of the *attachments* attribute. This attribute is a dictionary mapping attachment names to the attachment values. The attachment values are also a dictionary mapping attachment identifiers to their string values. Another quick example:

```
from edxml import EDXMLEvent

# Create an event
event = EDXMLEvent(properties={'names': {'Alice'}})

# Add a 'document' attachment with its SHA1 hash as ID
event.attachments['document'] = 'FooBar'

# Add a 'document' attachment while explicitly setting its ID
event.attachments['document'] = {'1': 'FooBar'}
```

As you can see in the above example, explicitly setting the identifiers of individual attachment values is not needed. When omitted, the SHA1 hashes of the attachment values will be used as identifiers.

3.5.3 EDXMLEvent Subclasses

The *EDXMLEvent* class has two subclasses. The first one is the *ParsedEvent* class. As the name suggests, this class is instantiated by EDXML parsers. In fact, it can only be instantiated by lxml, which is the library that the EDXML parser is built on. Its instances are a mix of a regular *EDXMLEvent* and a etree.Element instance. The reason for a separate parsed event variant is performance: The lxml library can generate these objects at minimal cost and can be passed through to EDXML writers for re-serialization at minimal cost.

The second subclass of *EDXMLEvent* is *EventElement*. This class is a wrapper around an lxml `etree.Element` instance containing an `<event>` XML element, providing the same convenient access interface as its parent, *EDXMLEvent*. The *EventElement* is mainly used for generating events that are intended for feeding to EDXML writers.

3.5.4 Object Value Types

Since EDXML data is XML, all object values in an EDXML document are strings. As a result, the events that are generated by the parser will only contain values of type str. When writing object values into an event Python types other than str can be used. For example, writing a float into an event object is perfectly fine.

What happens when a non-string value is written into an event depends on the particular event implementation. The base class *EDXMLEvent* does not care about the values stored in its properties, until it needs to be converted into an *EventElement*. This happens when the event is written using a *transcoder* or using the `EDXMLWriter`. At that point any non-string values are converted into strings. If that fails, an exception is raised.

Instances of *EventElement* are both an EDXML event and an XML element and the conversion to strings happens immediately when a property is written. This means that, in general, writing an event property may raise an exception.

The EDXML event implementations can convert various Python types into strings. These types include float, bool, datetime, Decimal and IP (from IPy).

3.5.5 Illegal XML characters

Some types of characters are illegal in XML. For that reason writing an object value string into an event can raise a ValueError. Using the *replace_invalid_characters()* function illegal characters can be automatically replaced by replacement characters.

3.5.6 Class Documentation

The class documentation of the various event implementations can be found below.

- *EDXMLEvent*
- *ParsedEvent*
- *EventElement*

EDXMLEvent

class edxml.**EDXMLEvent**(*properties=None*, *event_type_name=None*, *source_uri=None*, *parents=None*, *attachments=None*, *foreign_attribs=None*)

Bases: collections.abc.MutableMapping

Class representing an EDXML event.

The event allows its properties to be accessed and set much like a dictionary:

Event['property-name'] = 'value'

Note: Properties are sets of object values. On assignment, single values are automatically wrapped into sets.

Creates a new EDXML event. The Properties argument must be a dictionary mapping property names to object values. Object values must be lists of one or multiple object values. Explicit parent hashes must be specified as hex encoded strings. Attachments must be specified as a dictionary mapping attachment names to attachment values. The attachment values are dictionaries mapping attachment identifiers to the actual attachment strings.

Parameters

- **properties** (*Optional[Dict[str,Set[str]]]*) – Dictionary of properties
- **event_type_name** (*Optional[str]*) – Name of the event type
- **source_uri** (*Optional[str]*) – Event source URI
- **parents** (*Optional[List[str]]*) – List of explicit parent hashes
- **attachments** (*Optional[Dict[str, Dict[str]]]*) – Event attachments dictionary
- **foreign_attribs** (*Optional[Dict[str, str]]*) –

Returns EDXMLEvent

replace_invalid_characters(*replace=True*)

Enables automatic replacement of invalid unicode characters with the unicode replacement character. This will be used to produce valid XML representations of events containing invalid unicode characters in their property objects or attachments.

Enabling this feature may be useful when dealing with broken input data that triggers an occasional ValueError. In stead of crashing, the invalid data will be automatically replaced.

Parameters **replace** (*bool*) –

Returns edxml.EDXMLEvent

properties

Class property storing the properties of the event.

Returns Event properties

Return type PropertySet

attachments
Class property storing the attachments of the event.

Returns Dict[str, str]

get_any (*property_name*, *default=None*)
Convenience method for fetching any of possibly multiple object values of a specific event property. If the requested property has no object values, the specified default value is returned in stead.

Parameters

- **property_name** (*string*) – Name of requested property
- **default** – Default return value

Returns:

get_element (*sort=False*)
Returns the event as XML element. When the sort parameter is set to True, the properties, attachments and event parents are sorted as required for obtaining the event in its normal form as defined in the EDXML specification.

Parameters **sort** (*bool*) – Sort element components

Returns

Return type etree.Element

copy ()
Returns a copy of the event.

Returns EDXMLEvent

classmethod create (*properties=None*, *event_type_name=None*, *source_uri=None*, *parents=None*, *attachments=None*)
Creates a new EDXML event. The Properties argument must be a dictionary mapping property names to object values. Object values may be single values or a list of multiple object values. Explicit parent hashes must be specified as hex encoded strings.

Attachments are specified by means of a dictionary mapping attachment names to strings.

Note: For a slight performance gain, use the EDXMLEvent constructor directly to create new events.

Parameters

- **properties** (*Optional[Dict[str,Union[str,List[str]]]]*) – Dictionary of properties
- **event_type_name** (*Optional[str]*) – Name of the event type
- **source_uri** (*Optional[str]*) – Event source URI
- **parents** (*Optional[List[str]]*) – List of explicit parent hashes
- **attachments** (*Optional[str]*) – Event attachments dictionary

Returns

Return type *EDXMLEvent*

get_type_name ()
Returns the name of the event type.

Returns The event type name

Return type str

get_source_uri()
Returns the URI of the event source.

Returns The source URI

Return type str

get_properties()
Returns a dictionary containing property names as keys. The values are lists of object values.

Returns Event properties

Return type PropertySet

get_parent_hashes()
Returns a list of sticky hashes of parent events. The hashes are hex encoded strings.

Returns List of parent hashes

Return type List[str]

get_attachments()
Returns the attachments of the event as a dictionary mapping attachment names to the attachment values

Returns Event attachments

Return type Dict[str, str]

get_foreign_attributes()
Returns any non-edxml event attributes as a dictionary having the attribute names as keys and their associated values. The namespace is prepended to the keys in James Clark notation:

{'{http://some/foreign/namespace}attribute': 'value'

Returns: Dict[str, str]

set_properties(*properties*)
Replaces the event properties with the properties from specified dictionary. The dictionary must contain property names as keys. The values must be lists of strings.

Parameters **properties** – Dict(str, List(str)): Event properties

Returns

Return type *EDXMLEvent*

copy_properties_from(*source_event*, *property_map*)
Copies properties from another event, mapping property names according to specified mapping. The property_map argument is a dictionary mapping property names from the source event to property names in the target event, which is the event that is used to call this method.

If multiple source properties map to the same target property, the objects of both properties will be combined in the target property.

Parameters

- **source_event** (EDXMLEvent) –
- **property_map** (*dict(str,str)*) –

Returns

Return type *EDXMLEvent*

move_properties_from(*source_event*, *property_map*)
Moves properties from another event, mapping property names according to specified mapping. The property_map argument is a dictionary mapping property names from the source event to property names in the target event, which is the event that is used to call this method.

If multiple source properties map to the same target property, the objects of both properties will be combined in the target property.

Parameters

- **source_event** (EDXMLEvent) –
- **property_map** (*dict(str,str)*) –

Returns

Return type *EDXMLEvent*

set_type(*event_type_name*)
Set the event type.

Parameters **event_type_name** (*str*) – Name of the event type

Returns

Return type *EDXMLEvent*

set_attachment(*name*, *attachment*)
Set the event attachment associated with the specified name in the event type definition. The attachment argument accepts a string value. Alternatively a list can be given, allowing for multi-valued attachments. In that case, each attachment will have its SHA1 hash as unique identifier. Lastly, the attachment can be specified as a dictionary containing attachment identifiers as keys and the attachment strings as values. This allows control over choosing attachment identifiers.

Specifying None as attachment value removes the attachment from the event.

Parameters

- **name** (*str*) – Associated name in event type definition
- **attachment** (*Union[Optional[str], List[Optional[str]], Dict[str, Optional[str]]]*) – Attachment dictionary

Returns

Return type *EDXMLEvent*

set_source(*source_uri*)
Set the event source.

Parameters **source_uri** (*str*) – EDXML source URI

Returns

Return type *EDXMLEvent*

add_parents(*parent_hashes*)
Add the specified sticky hashes to the list of explicit event parents.

Parameters **parent_hashes** (*List[str]*) – list of sticky hash, as hexadecimal strings

Returns

Return type *EDXMLEvent*

set_parents(*parent_hashes*)
Replace the set of explicit event parents with the specified list of sticky hashes.

Parameters **parent_hashes** (*List[str]*) – list of sticky hash, as hexadecimal strings

Returns

Return type *EDXMLEvent*

set_foreign_attributes(*attribs*)

Sets foreign attributes. Foreign attributes are XML attributes not specified by EDXML and have a namespace that is not the EDXML namespace. The attributes can be passed as a dictionary. The keys in the dictionary must include the namespace in James Clark notation. Example:

{'{http://some/namespace}attribute_name': 'attribute_value'}

Parameters **attribs** (*Dict[str,str]*) – Attribute dictionary

Returns

Return type *EDXMLEvent*

compute_sticky_hash(*event_type*, *hash_function=<built-in function openssl_sha1>*, *encoding='hex'*)

Computes the sticky hash of the event. By default, the hash will be computed using the SHA1 hash function and encoded into a hexadecimal string. The hashing function can be adjusted to any of the hashing functions in the hashlib module. The encoding can be adjusted by setting the encoding argument to any string encoding that is supported by the str.encode() method.

Parameters

- **event_type** (edxml.ontology.EventType) – The event type
- **hash_function** (*callable*) – The hashlib hash function to use
- **encoding** (*str*) – Desired output encoding

Returns String representation of the hash.

Return type str

is_valid(*ontology*)

Check if an event is valid for a given ontology.

Parameters **ontology** (edxml.ontology.Ontology) – An EDXML ontology

Returns True if the event is valid

Return type bool

ParsedEvent

class edxml.**ParsedEvent**(*properties=None*, *event_type_name=None*, *source_uri=None*, *parents=None*, *attachments=None*, *foreign_attribs=None*)

Bases: edxml.event.EDXMLEvent, lxml.etree.ElementBase

This class extends both EDXMLEvent and etree.ElementBase to provide an EDXML event representation that can be generated directly by the lxml parser and can be treated much like it was a normal lxml Element representing an 'event' element

---

Note: The list and dictionary interfaces of etree.ElementBase are overridden by EDXMLEvent, so accessing keys will yield event properties rather than the XML attributes of the event element.

Note: This class can only be instantiated by parsers.

flush()
This class caches an alternative representation of the lxml Element, for internal use. Whenever the lxml Element is modified without using the dictionary interface, the flush() method must be called in order to refresh the internal state.

Returns

Return type *ParsedEvent*

copy()
Returns a copy of the event.

Returns

Return type *ParsedEvent*

classmethod create(*properties=None*, *event_type_name=None*, *source_uri=None*, *parents=None*, *attachments=None*)
This override of the create() method of the EDXMLEvent class only raises exceptions, because ParsedEvent objects can only be created by parsers.

Raises `NotImplementedError`

get_properties()
Returns a dictionary containing property names as keys. The values are lists of object values.

Returns Event properties

Return type PropertySet

get_attachments()
Returns the attachments of the event as a dictionary mapping attachment names to the attachment values

Returns Event attachments

Return type Dict[str, str]

get_foreign_attributes()
Returns any non-edxml event attributes as a dictionary having the attribute names as keys and their associated values. The namespace is prepended to the keys in James Clark notation:

{'{http://some/foreign/namespace}attribute': 'value'

Returns: Dict[str, str]

get_parent_hashes()
Returns a list of sticky hashes of parent events. The hashes are hex encoded strings.

Returns List of parent hashes

Return type List[str]

get_element(*sort=False*)
Returns the event as XML element. When the sort parameter is set to True, the properties, attachments and event parents are sorted as required for obtaining the event in its normal form as defined in the EDXML specification.

Parameters **sort** (*bool*) – Sort element components

Returns

Return type etree.Element

set_properties(*properties*)
Replaces the event properties with the properties from specified dictionary. The dictionary must contain property names as keys. The values must be lists of strings.

Parameters **properties** – Dict(str, List(str)): Event properties

Returns

Return type *EDXMLEvent*

set_attachment(*name*, *attachment*)
Set the event attachment associated with the specified name in the event type definition. The attachment argument accepts a string value. Alternatively a list can be given, allowing for multi-valued attachments. In that case, each attachment will have its SHA1 hash as unique identifier. Lastly, the attachment can be specified as a dictionary containing attachment identifiers as keys and the attachment strings as values. This allows control over choosing attachment identifiers.

Specifying None as attachment value removes the attachment from the event.

Parameters

- **name** (*str*) – Associated name in event type definition
- **attachment** (*Union[Optional[str], List[Optional[str]], Dict[str, Optional[str]]]*) – Attachment dictionary

Returns

Return type *ParsedEvent*

add_parents(*parent_hashes*)
Add the specified sticky hashes to the list of explicit event parents.

Parameters **parent_hashes** (*List[str]*) – list of sticky hashes, as hexadecimal strings

Returns

Return type *ParsedEvent*

set_parents(*parent_hashes*)
Replace the set of explicit event parents with the specified list of sticky hashes.

Parameters **parent_hashes** (*List[str]*) – list of sticky hashes, as hexadecimal strings

Returns

Return type *ParsedEvent*

set_foreign_attributes(*attribs*)
Sets foreign attributes. Foreign attributes are XML attributes not specified by EDXML and have a namespace that is not the EDXML namespace. The attributes can be passed as a dictionary. The keys in the dictionary must include the namespace in James Clark notation. Example:

{'{http://some/namespace}attribute_name': 'attribute_value'}

Parameters **attribs** (*Dict[str,str]*) – Attribute dictionary

Returns

Return type *EDXMLEvent*

get_type_name()
Returns the name of the event type.

Returns The event type name

Return type str

get_source_uri()
Returns the URI of the event source.

Returns The source URI

Return type str

set_type(*event_type_name*)
Set the event type.

Parameters **event_type_name** (*str*) – Name of the event type

Returns

Return type *EDXMLEvent*

set_source(*source_uri*)
Set the event source.

Parameters **source_uri** (*str*) – EDXML source URI

Returns

Return type *EDXMLEvent*

EventElement

class edxml.**EventElement**(*properties=None*, *event_type_name=None*, *source_uri=None*, *parents=None*, *attachments=None*, *foreign_attribs=None*)
Bases: edxml.event.EDXMLEvent

This class extends EDXMLEvent to provide an EDXML event representation that wraps an etree Element instance, providing a convenient means to generate and manipulate EDXML <event> elements. Using this class is preferred over using EDXMLEvent if you intend to feed it to EDXMLWriter.

Creates a new EDXML event. The Properties argument must be a dictionary mapping property names to object values. Object values must be lists of one or multiple strings. Explicit parent hashes must be specified as hex encoded strings. Attachments must be specified as a dictionary mapping attachment names to attachment values. The attachment values are dictionaries mapping attachment identifiers to the actual attachment strings.

Parameters

- **properties** (*Dict(str, List[str])*) – Dictionary of properties
- **event_type_name** (*Optional[str]*) – Name of the event type
- **source_uri** (*Optional[optional]*) – Event source URI
- **parents** (*Optional[List[str]]*) – List of explicit parent hashes
- **attachments** (*Optional[Dict[str, Dict[str, str]]]*) – Event attachments dictionary
- **foreign_attribs** (*Optional[Dict[str, str]]*) –

Returns

Return type *EventElement*

get_element(*sort=False*)
Returns the event as XML element. When the sort parameter is set to True, the properties, attachments and event parents are sorted as required for obtaining the event in its normal form as defined in the EDXML specification.

Parameters **sort** (*bool*) – Sort element components

Returns

Return type etree.Element

copy()
Returns a copy of the event.

Returns

Return type *EventElement*

classmethod create(*properties=None*, *event_type_name=None*, *source_uri=None*, *parents=None*, *attachments=None*)
Creates a new EDXML event. The Properties argument must be a dictionary mapping property names to object values. Object values may be single values or a list of multiple object values. Explicit parent hashes must be specified as hex encoded strings.

Note: For a slight performance gain, use the EventElement constructor directly to create new events.

Parameters

- **properties** (*Optional[Dict[str,Union[str,List[str]]]]*) – Dictionary of properties
- **event_type_name** (*Optional[str]*) – Name of the event type
- **source_uri** (*Optional[str]*) – Event source URI
- **parents** (*Optional[List[str]]*) – List of explicit parent hashes
- **attachments** (*Optional[Dict[str, Dict[str, str]]]*) – Event attachments dictionary

Returns

Return type *EventElement*

classmethod create_from_event(*event*)
Creates and returns a new EventElement instance by reading it from another EDXML event.

Parameters **event** (EDXMLEvent) – The EDXML event to copy data from

Returns

Return type *EventElement*

get_properties()
Returns a dictionary containing property names as keys. The values are lists of object values.

Returns Event properties

Return type PropertySet

get_attachments()
Returns the attachments of the event as a dictionary mapping attachment names to the attachment values

Returns Event attachments

Return type Dict[str, str]

get_foreign_attributes()
Returns any non-edxml event attributes as a dictionary having the attribute names as keys and their associated values. The namespace is prepended to the keys in James Clark notation:

{'{http://some/foreign/namespace}attribute': 'value'

Returns: Dict[str, str]

get_parent_hashes()
Returns a list of sticky hashes of parent events. The hashes are hex encoded strings.

Returns List of parent hashes

Return type List[str]

get_type_name()
Returns the name of the event type.

Returns The event type name

Return type str

get_source_uri()
Returns the URI of the event source.

Returns The source URI

Return type str

set_properties(*properties*)
Replaces the event properties with the properties from specified dictionary. The dictionary must contain property names as keys. The values must be lists of strings.

Parameters **properties** – Dict(str, List(str)): Event properties

Returns

Return type *EventElement*

set_attachment(*name*, *attachment*)
Set the event attachment associated with the specified name in the event type definition. The attachment argument accepts a string value. Alternatively a list can be given, allowing for multi-valued attachments. In that case, each attachment will have its SHA1 hash as unique identifier. Lastly, the attachment can be specified as a dictionary containing attachment identifiers as keys and the attachment strings as values. This allows control over choosing attachment identifiers.

Specifying None as attachment value removes the attachment from the event.

Parameters

- **name** (*str*) – Associated name in event type definition
- **attachment** (*Union[Optional[str], List[Optional[str]], Dict[str, Optional[str]]]*) – Attachment dictionary

Returns

Return type *EventElement*

add_parents(*parent_hashes*)
Add the specified sticky hashes to the list of explicit event parents.

Parameters **parent_hashes** (*List[str]*) – list of sticky hashes, as hexadecimal strings

Returns

Return type *EventElement*

set_parents(*parent_hashes*)
Replace the set of explicit event parents with the specified list of sticky hashes.

Parameters **parent_hashes** (*List[str]*) – list of sticky hashes, as hexadecimal strings

Returns

Return type *EventElement*

set_foreign_attributes(*attribs*)
Sets foreign attributes. Foreign attributes are XML attributes not specified by EDXML and have a namespace that is not the EDXML namespace. The attributes can be passed as a dictionary. The keys in the dictionary must include the namespace in James Clark notation. Example:

{'{http://some/namespace}attribute_name': 'attribute_value'}

Parameters **attribs** (*Dict[str,str]*) – Attribute dictionary

Returns

Return type *EDXMLEvent*

set_type(*event_type_name*)
Set the event type.

Parameters **event_type_name** (*str*) – Name of the event type

Returns

Return type *EDXMLEvent*

set_source(*source_uri*)
Set the event source.

Parameters **source_uri** (*str*) – EDXML source URI

Returns

Return type *EDXMLEvent*

3.6 EDXML Ontologies

When reading an EDXML data stream, the parser consumes both events and ontology information and provides an interface to access both. On the other hand, EDXML writers require providing an ontology matching the events that are written. In the EDXML SDK, ontology information is represented by instances of the *Ontology* class. This class is supported by several peripheral classes representing *event types*, *event properties*, *object types*, *data types* and *event sources*.

Ontologies can be populated by using the various named constructors of the *Ontology* class. Alternatively, ontology elements can be fetched from an *Ontology Brick*. or generated using an *event type factory*.

3.6.1 Ontology Description & Visualization

EDXML data sources can provide a wealth of knowledge. Learning if and how that knowledge will help you in your data analysis tasks requires studying its output ontology and how it fits into other domain ontologies that you might use. Unless an EDXML data source can just tell you.

The *describe_producer_rst()* function can do just that. Given an ontology it will list the object types and concepts that it uses and how it aids reasoning about data. For example, the description might say: *Identifies computers as network routers* or *Relates IP addresses to host names*. The name of the function indicates that the generated descriptions are meant for describing EDXML data producers such as data sources or data processors.

When designing an EDXML ontology, getting the relations and concept associations right is critical. A picture can be most helpful to review your design. Using the *generate_graph_property_concepts()* function you can generate a visualization showing the reasoning paths provided by a given ontology. It displays the concepts and object types and how they are connected.

3.6.2 API Documentation

The API documentation can be found below.

Ontology Elements

- *Ontology*
- *EventType*
- *EventTypeAttachment*
- *EventProperty*
- *PropertyConcept*
- *ObjectType*
- *Concept*
- *DataType*
- *EventSource*

Base Classes

- *OntologyElement*
- *VersionedOntologyElement*

Functions

- *generate_graph_property_concepts*
- *describe_producer_rst*

Ontology

class edxml.ontology.**Ontology**

Bases: edxml.ontology.ontology_element.OntologyElement

Class representing an EDXML ontology

clear()

Removes all event types, object types, concepts and event source definitions from the ontology.

Returns The ontology

Return type *edxml.ontology.Ontology*

get_version()

Returns the current ontology version. The initial version of a newly created empty ontology is zero. On each change, the version is incremented.

Note that this has nothing to do with versioning, upgrading and downgrading of EDXML ontologies. EDXML ontologies have no global version. The version that we return here is for change tracking.

Returns Ontology version

Return type int

is_modified_since(*version*)

Returns True if the ontology is newer than the specified version. Returns False if the ontology version is equal or older.

Returns

Return type bool

classmethod register_brick(*brick*)

Registers an ontology brick with the Ontology class, allowing Ontology instances to use any definitions offered by that brick. Ontology brick packages expose a register() method, which calls this method to register itself with the Ontology class.

Parameters **brick** (edxml.ontology.Brick) – Ontology brick

create_object_type(*name*, *display_name_singular=None*, *display_name_plural=None*, *description=None*, *data_type='string:0:mc:u'*)

Creates and returns a new ObjectType instance. When no display names are specified, display names will be created from the object type name. If only a singular form is specified, the plural form will be auto-generated by appending an 's'.

The object type is not validated on creation. This allows for creating a crude initial definition using this method and finish the definition later. If you do intend to create a valid definition from the start, it is recommended to validate it immediately.

Parameters

- **name** (*str*) – object type name
- **display_name_singular** (*str*) – display name (singular form)
- **display_name_plural** (*str*) – display name (plural form)
- **description** (*str*) – short description of the object type
- **data_type** (*str*) – a valid EDXML data type

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

create_concept(*name*, *display_name_singular=None*, *display_name_plural=None*, *description=None*)

Creates and returns a new Concept instance. When no display names are specified, display names will be created from the concept name. If only a singular form is specified, the plural form will be auto-generated by appending an 's'.

The concept is not validated on creation. This allows for creating a crude initial definition using this method and finish the definition later. If you do intend to create a valid definition from the start, it is recommended to validate it immediately.

Parameters

- **name** (*str*) – concept name
- **display_name_singular** (*str*) – display name (singular form)
- **display_name_plural** (*str*) – display name (plural form)
- **description** (*str*) – short description of the concept

Returns The Concept instance

Return type *edxml.ontology.Concept*

create_event_type(*name*, *display_name_singular=None*, *display_name_plural=None*, *description=None*)

Creates and returns a new EventType instance. When no display names are specified, display names will be created from the event type name. If only a singular form is specified, the plural form will be auto-generated by appending an 's'.

The event type is not validated on creation. This allows for creating a crude initial definition using this method and finish the definition later. If you do intend to create a valid definition from the start, it is recommended to validate it immediately.

Parameters

- **name** (*str*) – Event type name
- **display_name_singular** (*str*) – Display name (singular form)
- **display_name_plural** (*str*) – Display name (plural form)
- **description** (*str*) – Event type description

Returns The EventType instance

Return type *edxml.ontology.EventType*

create_event_source(*uri*, *description='no description available'*, *acquisition_date=None*)

Creates a new event source definition.

The source is not validated on creation. This allows for creating a crude initial definition using this method and finish the definition later. If you do intend to create a valid definition from the start, it is recommended to validate it immediately.

If the URI is missing a leading and / or trailing slash, these will be appended automatically.

Parameters

- **uri** (*str*) – The source URI
- **description** (*str*) – Description of the source
- **acquisition_date** (*Optional[str]*) – Acquisition date in format yyyymmdd

Returns

Return type *edxml.ontology.EventSource*

delete_object_type(*object_type_name*)

Deletes specified object type from the ontology, if it exists.

> **Warning:** Deleting object types may result in an invalid ontology.

Parameters **object_type_name** (*str*) – An EDXML object type name

Returns The ontology

Return type *edxml.ontology.Ontology*

delete_concept(*concept_name*)

Deletes specified concept from the ontology, if it exists.

> **Warning:** Deleting concepts may result in an invalid ontology.

Parameters **concept_name** (*str*) – An EDXML concept name

Returns The ontology

Return type *edxml.ontology.Ontology*

delete_event_type(*event_type_name*)
Deletes specified event type from the ontology, if it exists.

> **Warning:** Deleting event types may result in an invalid ontology.

Parameters **event_type_name** (*str*) – An EDXML event type name

Returns The ontology

Return type *edxml.ontology.Ontology*

delete_event_source(*source_uri*)
Deletes specified event source definition from the ontology, if it exists.

Parameters **source_uri** (*str*) – An EDXML event source URI

Returns The ontology

Return type *edxml.ontology.Ontology*

get_event_types()
Returns a dictionary containing all event types in the ontology. The keys are the event type names, the values are EventType instances.

Returns EventType instances

Return type Dict[str, *edxml.ontology.EventType*]

get_object_types()
Returns a dictionary containing all object types in the ontology. The keys are the object type names, the values are ObjectType instances.

Returns ObjectType instances

Return type Dict[str, *edxml.ontology.ObjectType*]

get_concepts()
Returns a dictionary containing all concepts in the ontology. The keys are the concept names, the values are Concept instances.

Returns Concept instances

Return type Dict[str, *edxml.ontology.Concept*]

get_event_sources()
Returns a dictionary containing all event sources in the ontology. The keys are the event source URIs, the values are EventSource instances.

Returns EventSource instances

Return type Dict[str, *edxml.ontology.EventSource*]

get_event_type_names()
Returns the list of names of all defined event types.

Returns List of event type names

Return type List[str]

get_object_type_names()
Returns the list of names of all defined object types.

Returns List of object type names

Return type List[str]

get_event_source_uris()
Returns the list of URIs of all defined event sources.

Returns List of source URIs

Return type List[str]

get_concept_names()
Returns the list of names of all defined concepts.

Returns List of concept names

Return type List[str]

get_event_type(*name*)
Returns the EventType instance having specified event type name, or None if no event type with that name exists.

Parameters **name** (*str*) – Event type name

Returns The event type instance

Return type *edxml.ontology.EventType*

get_object_type(*name*, *import_brick=True*)
Returns the ObjectType instance having specified object type name, or None if no object type with that name exists.

When the ontology does not contain the requested concept it will attempt to find the concept in any registered ontology bricks and import it. This can be turned off by setting import_brick to False.

Parameters

- **name** (*str*) – Object type name
- **import_brick** (*bool*) – Brick import flag

Returns The object type instance

Return type *edxml.ontology.ObjectType*

get_concept(*name*, *import_brick=True*)
Returns the Concept instance having specified concept name, or None if no concept with that name exists.

When the ontology does not contain the requested concept it will attempt to find the concept in any registered ontology bricks and import it. This can be turned off by setting import_brick to False.

Parameters

- **name** (*str*) – Concept name
- **import_brick** (*bool*) – Brick import flag

Returns The Concept instance

Return type *edxml.ontology.Concept*

get_event_source(*uri*)
Returns the EventSource instance having specified event source URI, or None if no event source with that URI exists.

Parameters **uri** (*str*) – Event source URI

Returns The event source instance

Return type *edxml.ontology.EventSource*

validate()
Checks if the defined ontology is a valid EDXML ontology.

Raises EDXMLOntologyValidationError

Returns The ontology

Return type *edxml.ontology.Ontology*

classmethod create_from_xml(*ontology_element*)

Parameters **ontology_element** (*lxml.etree.Element*) –

Returns The ontology

Return type *edxml.ontology.Ontology*

update(*other_ontology*, *validate=True*)
Updates the ontology using the definitions contained in another ontology. The other ontology may be specified in the form of an Ontology instance or an lxml Element containing a full ontology element.

Parameters

- **other_ontology** (*Union[lxml.etree.Element,* edxml.ontology.Ontology*]*) –
- **validate** (*bool*) – Validate the resulting ontology

Raises EDXMLOntologyValidationError

Returns The ontology

Return type *edxml.ontology.Ontology*

generate_xml()
Generates an lxml etree Element representing the EDXML <ontology> tag for this ontology.

Returns The element

Return type etree.Element

EventType

class edxml.ontology.**EventType**(*ontology*, *name*, *display_name_singular=None*, *display_name_plural=None*, *description=None*, *summary='no description available'*, *story='no description available'*, *parent=None*)
Bases: edxml.ontology.ontology_element.VersionedOntologyElement, collections.abc.MutableMapping

Class representing an EDXML event type. The class provides access to event properties by means of a dictionary interface. For each of the properties there is a key matching the name of the event property, the value is the property itself.

relations
Iterable[PropertyRelation]

Type Returns

get_name()
Returns the event type name

Returns

Return type str

get_description()
Returns the event type description

Returns

Return type str

get_display_name_singular()
Returns the event type display name, in singular form.

Returns

Return type str

get_display_name_plural()
Returns the event type display name, in plural form.

Returns

Return type Optional[str]

get_timespan_property_name_start()
Returns the name of the property that defines the start of the time span of the events. Returns None when the start of the event time span is the smallest of the event timestamps.

Returns

Return type Optional[str]

get_timespan_property_name_end()
Returns the name of the property that defines the end of the time span of the events. Returns None when the end of the event time span is the largest of the event timestamps.

Returns

Return type str

get_version_property_name()
Returns the name of the property that defines the version of the events that is used to merge colliding events. Returns None when the event type does not define an event version.

Returns

Return type Optional[str]

get_sequence_property_name()
Returns the name of the property that defines the sequence numbers of the events. Returns None when the event type does not define a sequence number.

Returns

Return type Optional[str]

get_properties()
Returns a dictionary containing all properties of the event type. The keys in the dictionary are the property names, the values are the EDXMLProperty instances.

Returns Properties

Return type Dict[str,*EventProperty*]

get_hashed_properties()
Returns a dictionary containing all properties of the event type that are used to compute event hashes. The keys in the dictionary are the property names, the values are the EDXMLProperty instances.

Returns Properties

Return type Dict[str, *EventProperty*]

get_property_relations(*relation_type=None*, *source=None*, *target=None*)
Returns a dictionary containing the property relations that are defined in the event type. The keys are relation IDs that should be considered opaque.

Optionally, the relations can be filtered on type, source and target.

Parameters

- **relation_type** (*str*) – Relation type
- **source** (*str*) – Name of source property
- **target** (*str*) – Name of target property

Returns

Return type Dict[str,PropertyRelation]

get_attachment(*name*)
Returns the specified attachment definition.

Raises KeyError

Returns

Return type *EventTypeAttachment*

get_attachments()
Returns a dictionary containing the attachments that are defined for the event type. The keys are attachment IDs.

Returns

Return type Dict[str,*EventTypeAttachment*]

get_timespan_property_names()
Returns a tuple containing the names of the properties that determine the start and end of the event time spans, in that order. Note that either may be None.

Returns Tuple[Optional[str], Optional[str]]

is_timeless()
Returns True when the event type is timeless, which means that it has no properties that are associated with the datetime data type. Returns False otherwise.

Returns

Return type bool

is_timeful()
Returns True when the event type is timeful, which means that it has at least one property that is associated with the datetime data type. Returns False otherwise.

Returns

Return type bool

get_summary_template()
Returns the event summary template.

Returns

Return type str

get_story_template()
Returns the event story template.

Returns

Return type str

get_parent()
Returns the parent event type, or None if no parent has been defined.

Returns The parent event type

Return type *EventTypeParent*

get_version()
Returns the version of the source definition.

Returns

Return type int

create_property(*name*, *object_type_name*, *description=None*)
Create a new event property.

Note: The description should be really short, indicating which role the object has in the event type.

Parameters

- **name** (*str*) – Property name
- **object_type_name** (*str*) – Name of the object type
- **description** (*str*) – Property description

Returns The EventProperty instance

Return type *EventProperty*

add_property(*prop*)
Add specified property

Parameters **prop** (EventProperty) – EventProperty instance

Returns The EventType instance

Return type *edxml.ontology.EventType*

remove_property(*property_name*)
Removes specified property from the event type.

Notes

Since the EventType class has a dictionary interface for accessing event type properties, you can also use the del operator to delete a property.

Parameters **property_name** (*str*) – The name of the property

Returns The EventType instance

Return type *edxml.ontology.EventType*

create_relation(*relation_type*, *source*, *target*, *description=None*, *predicate=None*, *source_concept_name=None*, *target_concept_name=None*, *confidence=None*)

Create a new property relation

Parameters

- **relation_type** (*str*) – Relation relation_type ('inter', 'intra', ...)
- **source** (*str*) – Name of source property
- **target** (*str*) – Name of target property
- **description** (*Optional[str]*) – Relation description, with property placeholders
- **predicate** (*Optional[str]*) – free form predicate
- **source_concept_name** (*Optional[str]*) – Name of the source concept
- **target_concept_name** (*Optional[str]*) – Name of the target concept
- **confidence** (*Optional[float]*) – Relation confidence [0.0,1.0]

Returns The PropertyRelation instance

Return type PropertyRelation

add_relation(*relation*)

Add specified property relation. It is recommended to use the methods from the EventProperty class in stead, to create property relations using a syntax that yields more readable code.

Parameters **relation** (*PropertyRelation*) – Property relation

Returns The EventType instance

Return type *edxml.ontology.EventType*

create_attachment(*name*)

Create a new attachment and add it to the event type. The description and singular display name are set to the attachment name. The plural form of the display name is constructed by appending an 's' to the singular form.

Parameters **name** (*str*) – attachment name

Returns EventTypeAttachment

add_attachment(*attachment*)

Add specified attachment definition to the event type.

Parameters **attachment** (EventTypeAttachment) – attachment definition

Returns The EventType instance

Return type *edxml.ontology.EventType*

make_child(*siblings_description*, *parent*)
Marks this event type as child of the specified parent event type. In case all hashed properties of the parent also exist in the child, a default property mapping will be generated, mapping properties based on identical property names.

Notes

You must call is_parent() on the parent before calling make_children()

Parameters

- **siblings_description** (*str*) – EDXML siblings-description attribute
- **parent** (edxml.ontology.EventType) – Parent event type

Returns The event type parent definition

Return type *EventTypeParent*

make_parent(*parent_description*, *child*)
Marks this event type as parent of the specified child event type.

Notes

To be used in conjunction with the make_children() method.

Parameters

- **parent_description** (*str*) – EDXML parent-description attribute
- **child** (edxml.ontology.EventType) – Child event type

Returns The EventType instance

Return type *edxml.ontology.EventType*

set_description(*description*)
Sets the event type description

Parameters **description** (*str*) – Description

Returns The EventType instance

Return type *edxml.ontology.EventType*

set_parent(*parent*)
Set the parent event type

Notes

It is recommended to use the make_children() and is_parent() methods in stead whenever possible, which results in more readable code.

Parameters **parent** (EventTypeParent) – Parent event type

Returns The EventType instance

Return type *edxml.ontology.EventType*

set_name(*event_type_name*)
Sets the name of the event type.

Parameters **event_type_name** (*str*) – Event type name

Returns The EventType instance

Return type *edxml.ontology.EventType*

set_display_name(*singular*, *plural=None*)
Configure the display name. If the plural form is omitted, it will be auto-generated by appending an 's' to the singular form.

Parameters

- **singular** (*str*) – Singular display name
- **plural** (*str*) – Plural display name

Returns The EventType instance

Return type *edxml.ontology.EventType*

set_summary_template(*summary*)
Set the event summary template

Parameters **summary** (*str*) – The event summary template

Returns The EventType instance

Return type *edxml.ontology.EventType*

set_story_template(*story*)
Set the event story template.

Parameters **story** (*str*) – The event story template

Returns The EventType instance

Return type *edxml.ontology.EventType*

set_version(*version*)
Sets the concept version

Parameters **version** (*int*) – Version

Returns The Concept instance

Return type *edxml.ontology.Concept*

set_timespan_property_name_start(*property_name*)
Sets the name of the property that defines the start of the time spans of the events.

Parameters **property_name** (*str*) –

Returns The EventType instance

Return type *edxml.ontology.EventType*

set_timespan_property_name_end(*property_name*)
Sets the name of the property that defines the end of the time spans of the events.

Parameters **property_name** (*str*) –

Returns The EventType instance

Return type *edxml.ontology.EventType*

set_version_property_name(*property_name*)
Sets the name of the property that defines the versions of the events that is used to merge colliding events.

Parameters **property_name** (*str*) –

Returns The EventType instance

Return type *edxml.ontology.EventType*

set_sequence_property_name(*property_name*)
Sets the name of the property that defines the sequence numbers of the events.

Parameters **property_name** (*str*) –

Returns The EventType instance

Return type *edxml.ontology.EventType*

evaluate_template(*edxml_event*, *which='story'*, *capitalize=True*, *colorize=False*)
Evaluates the event story or summary template of an event type using specified event, returning the result.

By default, the story template is evaluated, unless which is set to 'summary'.

By default, we will try to capitalize the first letter of the resulting string, unless capitalize is set to False.

Optionally, the output can be colorized. At his time this means that, when printed on the terminal, the objects in the evaluated string will be displayed using bold white characters.

Parameters

- **edxml_event** (edxml.EDXMLEvent) – the EDXML event to use
- **which** (*bool*) – which template to evaluate
- **capitalize** (*bool*) – Capitalize output or not
- **colorize** (*bool*) – Colorize output or not

Returns

Return type str

validate()
Checks if the event type definition is valid. Since it does not have access to the full ontology, the context of the event type is not considered. For example, it does not check if the event type definition refers to a parent event type that actually exists. Also, templates are not validated.

Raises EDXMLOntologyValidationError

Returns The EventType instance

Return type *EventType*

update(*event_type*)
Updates the event type to match the EventType instance passed to this method, returning the updated instance.

Parameters **event_type** (edxml.ontology.EventType) – The new EventType instance

Returns The updated EventType instance

Return type *edxml.ontology.EventType*

generate_xml()
Generates an lxml etree Element representing the EDXML <event-type> tag for this event type.

Returns The element

Return type etree.Element

get_singular_property_names()
Returns a list of properties that cannot have multiple values.

Returns List of property names

Return type list(str)

get_mandatory_property_names()
Returns a list of properties that must have a value

Returns List of property names

Return type list(str)

validate_event_structure(*edxml_event*)
Validates the structure of the event by comparing its properties and their object count to the requirements of the event type. Generates exceptions that are much more readable than standard XML validation exceptions.

Parameters **edxml_event** (edxml.EDXMLEvent) –

Raises EDXMLEventValidationError

Returns The EventType instance

Return type *edxml.ontology.EventType*

validate_event_objects(*event*, *property_name=None*)
Validates the object values in the event by comparing the values with their data types. Generates exceptions that are much more readable than standard XML validation exceptions.

Optionally the validation can be limited to a specific property only by setting the property_name argument.

Parameters

- **event** (edxml.EDXMLEvent) –
- **property_name** (*str*) –

Raises EDXMLEventValidationError

Returns The EventType instance

Return type *edxml.ontology.EventType*

validate_event_attachments(*event*, *attachment_name=None*)
Validates the attachment values in the event by comparing to the event type definition. Generates exceptions that are much more readable than standard XML validation exceptions.

Optionally the validation can be limited to a specific attachment only by setting the attachment_name argument.

Parameters

- **event** (edxml.EDXMLEvent) –
- **attachment_name** (*str*) –

Raises EDXMLEventValidationError

Returns The EventType instance

Return type *edxml.ontology.EventType*

normalize_event_objects(*event*, *property_names*)
Normalizes the object values in the event, resulting in valid EDXML object value strings. Raises an exception in case an object value cannot be normalized.

Parameters

- **event** (edxml.EDXMLEvent) –

- **property_names** (*List[str]*) –

Raises EDXMLEventValidationError

Returns The EventType instance

Return type *edxml.ontology.EventType*

generate_relax_ng(*ontology*, *namespaced=True*)

Returns an ElementTree containing a RelaxNG schema for validating events of this event type. It requires an Ontology instance for obtaining the definitions of objects types referred to by the properties of the event type.

By default, the schema expects the events to be namespaced. This can be turned off for validating events that will be added into an EDXML data stream that has a default namespace that events will inherit.

Parameters

- **ontology** (Ontology) – Ontology containing the event type
- **namespaced** (*bool*) – Require a namespace specification or not

Returns The schema

Return type lxml.etree.RelaxNG

merge_events(*events*)

Merges the specified events and returns the merged event. The merged event is an instance of the same class as the first input event.

Parameters **events** (*List[edxml.EDXMLEvent]*) – List of events

Returns Merged event

Return type *edxml.EDXMLEvent*

EventTypeAttachment

class edxml.ontology.**EventTypeAttachment**(*event_type*, *name='default'*, *media_type='text/plain'*, *display_name_singular=None*, *display_name_plural=None*, *description=None*, *encode_base64=False*)

Bases: edxml.ontology.ontology_element.OntologyElement

Class representing an EDXML event attachment definition

Creates a new event attachment definition.

When no description is given, the name is used as description. When no singular display name is given, the name is used as singular display name. When no plural display name is given, it is constructed by appending an 's' to the singular form.

Parameters

- **event_type** (edxml.ontology.EventType) – The event type containing the attachment definition
- **name** (*str*) – Name of the attachment
- **media_type** (*str*) – RFC 6838 media type
- **display_name_singular** (*str*) – display name (singular)
- **display_name_plural** (*str*) – display name (plural)

- **description** (*str*) – Description (EDXML template)
- **encode_base64** (*bool*) – Encode as base64 string yes / no

set_description(*description*)
Sets the EDXML attachment description

Parameters **description** (*str*) – description

Returns The EventTypeAttachment instance

Return type *edxml.ontology.EventTypeAttachment*

set_display_name(*singular*, *plural=None*)
Configure the display name. If the plural form is omitted, it will be auto-generated by appending an 's' to the singular form.

Parameters

- **singular** (*str*) – Singular display name
- **plural** (*str*) – Plural display name

Returns The EventTypeAttachment instance

Return type *edxml.ontology.EventTypeAttachment*

set_media_type(*media_type*)
Configure the media type. This must be a valid RFC 6838 media type.

Parameters **media_type** (*str*) – Media type

Returns The EventTypeAttachment instance

Return type *edxml.ontology.EventTypeAttachment*

set_encoding(*encoding*)
Sets the encoding to either 'unicode' or 'base64'.

Returns The EventTypeAttachment instance

Return type *edxml.ontology.EventTypeAttachment*

set_encoding_unicode()
Sets the encoding to unicode, which means that the attachment must be a valid unicode string. This is the default encoding for attachments.

Returns The EventTypeAttachment instance

Return type *edxml.ontology.EventTypeAttachment*

set_encoding_base64()
Sets the encoding to base64, which means that the attachment must be a valid base64 encoding string.

Returns The EventTypeAttachment instance

Return type *edxml.ontology.EventTypeAttachment*

get_name()
Returns the attachment name

Returns

Return type str

get_description()
Returns the attachment description

Returns

Return type str

get_display_name_singular()
Returns the display name, in singular form.

Returns

Return type str

get_display_name_plural()
Returns the display name, in plural form.

Returns

Return type str

get_media_type()
Returns the media type.

Returns

Return type str

get_encoding()
Returns the encoding, either 'unicode' or 'base64'.

Returns

Return type str

is_unicode_string()
Returns True when the attachment is a unicode encoded string, returns False otherwise.

Returns

Return type bool

is_base64_string()
Returns True when the attachment is a base64 encoded string, returns False otherwise.

Returns

Return type bool

validate()
Checks if the event type attachment is valid. It only looks at the attributes of the definition itself. For example, it does not check that the attachment has an id that is unique within the event type.

Raises EDXMLOntologyValidationError

Returns The EventTypeAttachment instance

Return type *edxml.ontology.EventTypeAttachment*

update(*attachment*)
Updates the attachment to match the EventTypeAttachment instance passed to this method, returning the updated instance.

Parameters **attachment** (edxml.ontology.EventTypeAttachment) – The new EventTypeAttachment instance

Returns The updated EventTypeAttachment instance

Return type *edxml.ontology.EventTypeAttachment*

generate_xml()
Generates an lxml etree Element representing the EDXML <attachment> tag for this event type attachment.

Returns The element

Return type etree.Element

EventTypeParent

class edxml.ontology.**EventTypeParent**(*child_event_type*, *parent_event_type_name*, *property_map*, *parent_description='belonging to'*, *siblings_description='sharing'*)
Bases: edxml.ontology.ontology_element.OntologyElement

Class representing an EDXML event type parent

classmethod create(*child_event_type*, *parent_event_type_name*, *property_map*, *parent_description='belonging to'*, *siblings_description='sharing'*)
Creates a new event type parent. The PropertyMap argument is a dictionary mapping property names of the child event type to property names of the parent event type.

Note: All hashed properties of the parent event type must appear in the property map.

Note: The parent event type must be defined in the same EDXML stream as the child.

Parameters

- **child_event_type** (EventType) – The child event type
- **parent_event_type_name** (*str*) – Name of the parent event type
- **property_map** (*Dict[str, str]*) – Property map
- **parent_description** (*Optional[str]*) – The EDXML parent-description attribute
- **siblings_description** (*Optional[str]*) – The EDXML siblings-description attribute

Returns The EventTypeParent instance

Return type *edxml.ontology.EventTypeParent*

set_parent_description(*description*)
Sets the EDXML parent-description attribute

Parameters **description** (*str*) – The EDXML parent-description attribute

Returns The EventTypeParent instance

Return type *edxml.ontology.EventTypeParent*

set_siblings_description(*description*)
Sets the EDXML siblings-description attribute

Parameters **description** (*str*) – The EDXML siblings-description attribute

Returns The EventTypeParent instance

Return type *edxml.ontology.EventTypeParent*

map(*child_property_name*, *parent_property_name=None*)
Add a property mapping, mapping a property in the child event type to the corresponding property in the parent. When the parent property name is omitted, it is assumed that the parent and child properties are named identically.

Parameters

- **child_property_name** (*str*) – Child property
- **parent_property_name** (*str*) – Parent property

Returns The EventTypeParent instance

Return type *edxml.ontology.EventTypeParent*

get_event_type_name()
Returns the name of the parent event type.

Returns

Return type str

get_property_map()
Returns the property map as a dictionary mapping property names of the child event type to property names of the parent.

Returns

Return type Dict[str,str]

get_parent_description()
Returns the EDXML 'parent-description' attribute.

Returns

Return type str

get_siblings_description()
Returns the EDXML 'siblings-description' attribute.

Returns

Return type str

validate()
Checks if the event type parent is valid. It only looks at the attributes of the definition itself. Since it does not have access to the full ontology, the context of the parent is not considered. For example, it does not check if the parent definition refers to an event type that actually exists.

Raises EDXMLOntologyValidationError

Returns The EventTypeParent instance

Return type *edxml.ontology.EventTypeParent*

update(*parent*)
Updates the event type parent to match the EventTypeParent instance passed to this method, returning the updated instance.

Parameters **parent** (edxml.ontology.EventTypeParent) – The new EventTypeParent instance

Returns The updated EventTypeParent instance

Return type *edxml.ontology.EventTypeParent*

generate_xml()
Generates an lxml etree Element representing the EDXML <parent> tag for this event type parent.

Returns The element

Return type etree.Element

EventProperty

class edxml.ontology.**EventProperty**(*event_type*, *name*, *object_type*, *description=None*, *optional=False*, *multivalued=False*, *merge='any'*, *similar=''*, *confidence=10*)
Bases: edxml.ontology.ontology_element.OntologyElement

Class representing an EDXML event property

MERGE_MATCH = 'match'
Merge strategy 'match'

MERGE_ANY = 'any'
Merge strategy 'any'

MERGE_ADD = 'add'
Merge strategy 'add'

MERGE_SET = 'set'
Merge strategy 'set'

MERGE_REPLACE = 'replace'
Merge strategy 'replace'

MERGE_MIN = 'min'
Merge strategy 'min'

MERGE_MAX = 'max'
Merge strategy 'max'

get_name()
Returns the property name.

Returns

Return type str

get_description()
Returns the property description.

Returns

Return type str

get_object_type_name()
Returns the name of the associated object type.

Returns

Return type str

get_merge_strategy()
Returns the merge strategy.

Returns

Return type str

get_similar_hint()
Get the EDXML 'similar' attribute.

Returns

Return type str

get_confidence()
Returns the property confidence.

Returns

Return type int

get_object_type()
Returns the ObjectType instance that is associated with the property.

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

get_data_type()
Returns the DataType instance that is associated with the object type of the property.

Returns The DataType instance

Return type *edxml.ontology.DataType*

get_concept_associations()
Returns a dictionary containing the names of all associated concepts as keys and the PropertyConcept instances as values.

Returns

Return type Dict[str,*edxml.ontology.PropertyConcept*]

relate_to(*type_predicate*, *target_property_name*, *reason=None*, *confidence=10*)
Creates and returns a relation between this property and the specified target property.

When no reason is specified, the reason is constructed by wrapping the type predicate with the place holders for the two properties.

Parameters

- **type_predicate** (*str*) – free form predicate
- **target_property_name** (*str*) – Name of the related property
- **reason** (*str*) – Relation description, with property placeholders
- **confidence** (*int*) – Relation confidence [00,10]

Returns The EventPropertyRelation instance

Return type edxml.ontology.EventPropertyRelation

relate_inter(*type_predicate*, *target_property_name*, *source_concept_name=None*, *target_concept_name=None*, *reason=None*, *confidence=10*)
Creates and returns a relation between this property and the specified target property. The relation is an 'inter' relation, indicating that the related objects belong to different, related concept instances.

When any of the related properties is associated with more than one concept, you are required to specify which of the associated concepts is involved in the relation.

When no reason is specified, the reason is constructed by wrapping the type predicate with the place holders for the two properties.

Parameters

- **type_predicate** (*str*) – free form predicate
- **target_property_name** (*str*) – Name of the related property
- **source_concept_name** (*str*) – Name of the source concept
- **target_concept_name** (*str*) – Name of the target concept
- **reason** (*str*) – Relation description, with property placeholders
- **confidence** (*int*) – Relation confidence [0,10]

Returns The EventPropertyRelation instance

Return type edxml.ontology.EventPropertyRelation

relate_intra(*type_predicate*, *target_property_name*, *source_concept_name=None*, *target_concept_name=None*, *reason=None*, *confidence=10*)

Creates and returns a relation between this property and the specified target property. The relation is an 'intra' relation, indicating that the related objects belong to the same concept instance.

When no reason is specified, the reason is constructed by wrapping the type predicate with the place holders for the two properties.

Parameters

- **target_property_name** (*str*) – Name of the related property
- **source_concept_name** (*str*) – Name of the source concept
- **target_concept_name** (*str*) – Name of the target concept
- **reason** (*str*) – Relation description, with property placeholders
- **type_predicate** (*str*) – free form predicate
- **confidence** (*float*) – Relation confidence [0,10]

Returns The EventPropertyRelation instance

Return type edxml.ontology.EventPropertyRelation

relate_name(*target_property_name*)

Creates and returns a relation between this property and the specified target property. The relation is a 'name' relation, indicating that the property provides names for the values of the target property.

Parameters **target_property_name** (*str*) – Name of the related property

Returns The EventPropertyRelation instance

Return type edxml.ontology.EventPropertyRelation

relate_description(*target_property_name*)

Creates and returns a relation between this property and the specified target property. The relation is a 'description' relation, indicating that the property provides descriptions for the values of the target property.

Parameters **target_property_name** (*str*) – Name of the related property

Returns The EventPropertyRelation instance

Return type edxml.ontology.EventPropertyRelation

relate_container(*target_property_name*)

Creates and returns a relation between this property and the specified target property. The relation is a 'container' relation, indicating that the values of the property contain the values of the target property. In other words, the values of the target property are conceptually a part of the values of this property.

Parameters **target_property_name** (*str*) – Name of the related property

Returns The EventPropertyRelation instance

Return type edxml.ontology.EventPropertyRelation

relate_original(*target_property_name*)
Creates and returns a relation between this property and the specified target property. The relation is an 'original' relation, indicating that the value of the property contains the original version of the value of the target property.

Parameters **target_property_name** (*str*) – Name of the related property

Returns The EventPropertyRelation instance

Return type edxml.ontology.EventPropertyRelation

add_associated_concept(*concept_association*)
Add the specified concept association.

Parameters **concept_association** (edxml.ontology.PropertyConcept) – Property concept association

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

set_merge_strategy(*merge_strategy*)
Set the merge strategy of the property. This should be one of the MERGE_* attributes of this class.

Automatically makes the property mandatory or single valued when the merge strategy requires it.

Parameters **merge_strategy** (*str*) – The merge strategy

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

set_description(*description*)
Set the description of the property. This should be really short, indicating which role the object has in the event type.

Parameters **description** (*str*) – The property description

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

set_confidence(*confidence*)
Configure the property confidence

Parameters **confidence** (*int*) – Property confidence [1,10]

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

make_optional()
Make the property optional.

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

make_mandatory()
Make the property mandatory.

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

set_optional(*is_optional*)
Set the optional flag for the property to True (property is optional) or False (property is mandatory).

Parameters **is_optional** (*bool*) –

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

make_single_valued()
Make the property single-valued.

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

make_multivalued()
Make the property multi-valued.

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

make_hashed()
Changes the property into a hashed property by setting its merge strategy to 'match'.

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

is_hashed()
Returns True if property has merge strategy 'match', which means that it is included in event hashes

Returns

Return type bool

is_optional()
Returns True if property is optional, returns False otherwise

Returns

Return type bool

is_mandatory()
Returns True if property is mandatory, returns False otherwise

Returns

Return type bool

is_multi_valued()
Returns True if property is multi-valued, returns False otherwise

Returns

Return type bool

is_single_valued()
Returns True if property is single-valued, returns False otherwise

Returns

Return type bool

identifies(*concept_name*, *confidence=10*, *cnp=128*)
Marks the property as an identifier for specified concept, with specified confidence.

Parameters

- **concept_name** (*str*) – concept name
- **confidence** (*int*) – concept identification confidence [0, 10]
- **cnp** (*int*) – concept naming priority [0,255]

Returns The PropertyConcept association

Return type *edxml.ontology.PropertyConcept*

set_multi_valued(*is_multivalued*)
Configures the property as multi-valued or single-valued

Parameters **is_multivalued** (*bool*) –

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

hint_similar(*similarity*)
Set the EDXML 'similar' attribute.

Parameters **similarity** (*str*) – similar attribute string

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

merge_add()
Set merge strategy to 'add'.

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

merge_replace()
Set merge strategy to 'replace'.

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

merge_set()
Set merge strategy to 'set'.

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

merge_any()
Set merge strategy to 'any', which is the default merge strategy.

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

merge_min()
Set merge strategy to 'min'.

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

merge_max()
Set merge strategy to 'max'.

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

validate()
Checks if the property definition is valid. It only looks at the attributes of the property itself. Since it does not have access to the full ontology, the context of the property is not considered. For example, it does not check if the object type in the property actually exist.

Raises EDXMLOntologyValidationError

Returns The EventProperty instance

Return type *edxml.ontology.EventProperty*

update(*event_property*)
Updates the event property to match the EventProperty instance passed to this method, returning the updated instance.

Parameters **event_property** (edxml.ontology.EventProperty) – The new EventProperty instance

Returns The updated EventProperty instance

Return type *edxml.ontology.EventProperty*

generate_xml()
Generates an lxml etree Element representing the EDXML <property> tag for this event property.

Returns The element

Return type etree.Element

PropertyConcept

class edxml.ontology.**PropertyConcept**(*event_type*, *event_property*, *concept_name*, *confidence=10*, *naming_priority=128*)
Bases: edxml.ontology.ontology_element.OntologyElement

Class representing an association between a property and a concept

get_concept_name()
Returns the name of the associated concept.

Returns

Return type str

get_property_name()
Returns the name of the event property.

Returns

Return type str

get_confidence()
Returns the concept confidence of the association, indicating how strong the property values are as identifiers of instances of the associated concept.

Returns

Return type int

get_concept_naming_priority()
Returns the concept naming priority.

Returns

Return type int

get_attribute_name()
Returns the full name of the concept attribute, which includes the object type name, a colon and its extension.

Returns

Return type str

get_attribute_name_extension()
Returns the attribute name extension.

Returns

Return type str

get_attribute_display_name_singular()
Returns the display name of the concept attribute (singular form). When the attribute does not have a custom display name, the display name of the object type of the associated property is used.

Returns

Return type str

get_attribute_display_name_plural()
Returns the display name of the concept attribute (plural form) When the attribute does not have a custom display name, the display name of the object type of the associated property is used.

Returns

Return type str

set_confidence(*confidence*)
Configure the concept confidence of the association, indicating how strong the property values are as identifiers of instances of the associated concept.

Parameters **confidence** (*int*) – Confidence [1,10]

Returns The PropertyConcept instance

Return type *edxml.ontology.PropertyConcept*

set_concept_naming_priority(*priority*)
Configure the concept naming priority.

Parameters **priority** (*int*) – Naming priority [1,10]

Returns The PropertyConcept instance

Return type *edxml.ontology.PropertyConcept*

set_attribute(*extension*, *display_name_singular=''*, *display_name_plural=''*)
Set the name extension of the concept attribute. By default, concept attributes are named after the object type of their associated property by taking the object type name, appending a colon and appending a concept name extension.

If the plural form of the display name is omitted, it will be auto-generated by appending an 's' to the singular form.

Parameters

- **extension** (*str*) –
- **display_name_singular** (*str*) –
- **display_name_plural** (*str*) –

Returns The PropertyConcept instance

Return type *edxml.ontology.PropertyConcept*

validate()
Checks if the concept association is valid. It only looks at the attributes of the association itself. Since it does not have access to the full ontology, the context of the association is not considered. For example, it does not check if the concept actually exist.

Raises EDXMLOntologyValidationError

Returns The PropertyConcept instance

Return type *edxml.ontology.PropertyConcept*

update(*property_concept*)
Updates the concept association to match the PropertyConcept instance passed to this method, returning the updated instance.

Parameters **property_concept** (edxml.ontology.PropertyConcept) – The new PropertyConcept instance

Returns The updated PropertyConcept instance

Return type *edxml.ontology.PropertyConcept*

generate_xml()
Generates an lxml etree Element representing the EDXML <property-concept> tag for this property concept association

Returns The element

Return type etree.Element

ObjectType

class edxml.ontology.**ObjectType**(*ontology*, *name*, *display_name_singular=None*, *display_name_plural=None*, *description=None*, *data_type='string:0:mc:u'*, *unit_name=None*, *unit_symbol=None*, *prefix_radix=None*, *compress=False*, *xref=None*, *fuzzy_matching=None*, *regex_hard=None*, *regex_soft=None*)
Bases: edxml.ontology.ontology_element.VersionedOntologyElement

Class representing an EDXML object type

get_name()
Returns the name of the object type.

Returns The object type name

Return type str

get_display_name_singular()
Returns the display name of the object type, in singular form.

Returns

Return type str

get_display_name_plural()
Returns the display name of the object type, in plural form.

Returns

Return type str

get_description()
Returns the description of the object type.

Returns

Return type str

get_data_type()
Returns the data type of the object type.

Returns The data type

Return type *edxml.ontology.DataType*

get_unit_name()
Returns the name of the measurement unit or None in case the object type does not have any associated unit.

Returns unit name

Return type Optional[str]

get_unit_symbol()
Returns the symbol of the measurement unit or None in case the object type does not have any associated unit.

Returns unit symbol

Return type Optional[str]

get_prefix_radix()
Returns the natural radix that should be used for metric prefixes of numerical object types.

Returns radix

Return type Optional[int]

is_compressible()
Returns True if compression is advised for the object type, returns False otherwise.

Returns

Return type bool

get_xref()
Returns the external reference to additional information about the object type or None in case it does not define any.

Returns xref

Return type Optional[str]

get_fuzzy_matching()
Returns the EDXML fuzzy-matching attribute for the object type or None in case it does not define any.

Returns

Return type Optional[str]

get_regex_hard()
Returns the regular expression that object values must match. Returns None in case no hard regular expression is associated with the object type.

Note that the regular expression is not anchored to the start and end of the object value string even though object values must fully matches the expression from start to end. Be sure to wrap the expression in anchors where full string matching is needed.

Returns

Return type Optional[str]

get_regex_soft()
Returns the regular expression that can be used to identify valid object values or generate synthetic values. Returns None in case no soft regular expression is associated with the object type.

Note that the regular expression is not anchored to the start and end of the object value string even though the expression should match full object values from start to end. Be sure to wrap the expression in anchors before use.

Returns

Return type Optional[str]

get_version()
Returns the version of the source definition.

Returns

Return type int

set_description(*description*)
Sets the object type description

Parameters **description** (*str*) – Description

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

set_data_type(*data_type*)
Configure the data type.

Parameters **data_type** (edxml.ontology.DataType) – DataType instance

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

set_unit(*unit_name*, *unit_symbol*)
Configure the measurement unit name and symbol.

Parameters

- **unit_name** (*str*) – Unit name
- **unit_symbol** (*str*) – Unit symbol

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

set_prefix_radix(*radix*)
Configure the natural radix that should be used for metric prefixes of numerical object types

Parameters **radix** (*int*) – Radix

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

set_xref(*url*)
Configure the URL pointing to additional information about the object type.

Parameters **url** (*int*) – URL

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

set_display_name(*singular*, *plural=None*)
Configure the display name. If the plural form is omitted, it will be auto-generated by appending an 's' to the singular form.

Parameters

- **singular** (*str*) – display name (singular form)
- **plural** (*str*) – display name (plural form)

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

set_regex_hard(*pattern*)
Configure a regular expression that object values must match.

Parameters **pattern** (*str*) – Regular expression

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

set_regex_soft(*pattern*)
Configure a regular expression that should match values that are valid for the object type.

Parameters **pattern** (*str*) – Regular expression

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

set_fuzzy_matching_attribute(*attribute*)
Sets the EDXML fuzzy-matching attribute.

Notes

It is recommended to use the FuzzyMatch...() methods in stead to configure fuzzy matching.

Parameters **attribute** (*str*) – The attribute value

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

set_version(*version*)
Sets the object type version

Parameters **version** (*int*) – Version

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

upgrade()
Verifies if the current instance is a valid upgrade of the instance as it was when the version was last changed. When successful the version number is incremented.

This method is used for fluent upgrading of ontology bricks, allowing definitions of object types to be changed in a single call chain while making sure that no backward incompatible changes are made.

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

fuzzy_match_head(*length*)
Configure fuzzy matching on the head of the string (only for string data types).

Parameters **length** (*int*) – Number of characters to match

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

fuzzy_match_tail(*length*)
Configure fuzzy matching on the tail of the string (only for string data types).

Parameters **length** (*int*) – Number of characters to match

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

fuzzy_match_substring(*pattern*)
Configure fuzzy matching on a substring (only for string data types).

Parameters **pattern** (*str*) – Regular expression

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

fuzzy_match_phonetic()
Configure fuzzy matching on the sound of the string (phonetic fingerprinting).

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

compress(*is_compressible=True*)
Enable or disable compression for the object type.

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

validate_object_value(*value*)
Validates the provided object value against the object type definition as well as its data type, raising an EDXMLValidationException when the value is invalid.

Parameters **value** (*str*) – Object value

Raises EDXMLEventValidationError

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

validate()
Checks if the object type is valid. It only looks at the attributes of the definition itself. Since it does not have access to the full ontology, the context of the event type is not considered. For example, it does not check if other, conflicting object type definitions exist.

Raises EDXMLOntologyValidationError

Returns The ObjectType instance

Return type *edxml.ontology.ObjectType*

update(*object_type*)

Parameters **object_type** (edxml.ontology.ObjectType) – The new ObjectType instance

Returns The updated ObjectType instance

Return type *edxml.ontology.ObjectType*

generate_xml()

Generates an lxml etree Element representing the EDXML <object-type> tag for this object type.

Returns The element

Return type etree.Element

Concept

class edxml.ontology.**Concept**(*ontology*, *name*, *display_name_singular=None*, *display_name_plural=None*, *description=None*)

Bases: edxml.ontology.ontology_element.VersionedOntologyElement

Class representing an EDXML concept

classmethod concept_names_share_branch(*a*, *b*)

Returns True when concept name a is a specialization of b or the other way around. This is true when both share the same branch in the concept name hierarchy. For example concept names a.b and a.b.c share the same branch while a.b and a.c do not.

Parameters

- **a** (*str*) –
- **b** (*str*) –

Returns

Return type bool

classmethod concept_name_is_specialization(*concept_name*, *specialization_concept_name*)

Returns True when the one concept name a is a specialization of the other. This is true when the specialization extends the concept name by appending to it.

Parameters

- **concept_name** (*str*) –
- **specialization_concept_name** (*str*) –

Returns

Return type bool

get_name()

Returns the name of the concept.

Returns The concept name

Return type str

get_display_name_singular()

Returns the display name of the concept, in singular form.

Returns

Return type str

get_display_name_plural()
Returns the display name of the concept, in plural form.

Returns

Return type str

get_description()
Returns the description of the concept.

Returns

Return type str

get_version()
Returns the version of the concept definition.

Returns

Return type int

set_description(*description*)
Sets the concept description

Parameters **description** (*str*) – Description

Returns The Concept instance

Return type *edxml.ontology.Concept*

set_display_name(*singular*, *plural=None*)
Configure the display name. If the plural form is omitted, it will be auto-generated by appending an 's' to the singular form.

Parameters

- **singular** (*str*) – display name (singular form)
- **plural** (*str*) – display name (plural form)

Returns The Concept instance

Return type *edxml.ontology.Concept*

set_version(*version*)
Sets the concept version

Parameters **version** (*int*) – Version

Returns The Concept instance

Return type *edxml.ontology.Concept*

upgrade()
Verifies if the current instance is a valid upgrade of the instance as it was when the version was last changed. When successful the version number is incremented.

This method is used for fluent upgrading of ontology bricks, allowing definitions of concepts to be changed in a single call chain while making sure that no backward incompatible changes are made.

Returns The Concept instance

Return type *edxml.ontology.Concept*

validate()
Checks if the concept is valid. It only looks at the attributes of the definition itself. Since it does not have access to the full ontology, the context of the ontology is not considered. For example, it does not check if other, conflicting concept definitions exist.

Raises EDXMLOntologyValidationError

Returns The Concept instance

Return type *edxml.ontology.Concept*

update(*concept*)
Update the concept using information from the provided concept and validate the result.

Parameters **concept** (edxml.ontology.Concept) – The new Concept instance

Returns The updated Concept instance

Return type *edxml.ontology.Concept*

generate_xml()
Generates an lxml etree Element representing the EDXML <concept> tag for this concept.

Returns The element

Return type etree.Element

classmethod generate_specializations(*concept_name*, *parent_concept_name=None*)
Generator yielding specializations of a given concept. For example, when given a concept name 'a.b.c' it will generate concept names 'a', 'a.b' and 'a.b.c'.

Optionally a parent concept can be specified. In that case the specializations will be generated starting at the parent concept.

Parameters

- **concept_name** (*str*) – Concept for which to generate specializations
- **parent_concept_name** (*Optional[str]*) – First yielded concept name

Yields str

classmethod generate_generalizations(*concept_name*)
Generator yielding generalizations of a given concept. For example, when given a concept name 'a.b.c' it will generate concept names 'a.b' and 'a'.

Parameters **concept_name** (*str*) – Concept for which to generate generalizations

Yields str

DataType

class edxml.ontology.**DataType**(*data_type*)
Bases: object

Class representing an EDXML data type. Instances of this class can be cast to strings, which yields the EDXML data-type attribute.

classmethod datetime()
Create a datetime DataType instance.

Returns

Return type *DataType*

classmethod sequence()
Create a sequence DataType instance.

Returns

Return type *DataType*

classmethod boolean()
Create a boolean value DataType instance.

Returns

Return type *edxml.ontology.DataType*

classmethod tiny_int(*signed=True*)
Create an 8-bit tinyint DataType instance.

Parameters **signed** (*bool*) – Create signed or unsigned number

Returns

Return type *edxml.ontology.DataType*

classmethod small_int(*signed=True*)
Create a 16-bit smallint DataType instance.

Parameters **signed** (*bool*) – Create signed or unsigned number

Returns

Return type *edxml.ontology.DataType*

classmethod medium_int(*signed=True*)
Create a 24-bit mediumint DataType instance.

Parameters **signed** (*bool*) – Create signed or unsigned number

Returns

Return type *edxml.ontology.DataType*

classmethod int(*signed=True*)
Create a 32-bit int DataType instance.

Parameters **signed** (*bool*) – Create signed or unsigned number

Returns

Return type *edxml.ontology.DataType*

classmethod big_int(*signed=True*)
Create a 64-bit bigint DataType instance.

Parameters **signed** (*bool*) – Create signed or unsigned number

Returns

Return type *edxml.ontology.DataType*

classmethod float(*signed=True*)
Create a 32-bit float DataType instance.

Parameters **signed** (*bool*) – Create signed or unsigned number

Returns

Return type *edxml.ontology.DataType*

classmethod double(*signed=True*)
Create a 64-bit double DataType instance.

Parameters **signed** (*bool*) – Create signed or unsigned number

Returns

Return type *edxml.ontology.DataType*

classmethod decimal(*total_digits*, *fractional_digits*, *signed=True*)
Create a decimal DataType instance.

Parameters

- **total_digits** (*int*) – Total number of digits
- **fractional_digits** (*int*) – Number of digits after the decimal point
- **signed** (*bool*) – Create signed or unsigned number

Returns

Return type *edxml.ontology.DataType*

classmethod currency()
Create a currency DataType instance.

Returns

Return type *edxml.ontology.DataType*

classmethod string(*length=0*, *lower_case=True*, *upper_case=True*, *require_unicode=True*, *reverse_storage=False*)
Create a string DataType instance.

Parameters

- **length** (*int*) – Max number of characters (zero = unlimited)
- **lower_case** (*bool*) – Allow lower case characters
- **upper_case** (*bool*) – Allow upper case characters
- **require_unicode** (*bool*) – String may contain UTF-8 characters
- **reverse_storage** (*bool*) – Hint storing the string in reverse character order

Returns

Return type *edxml.ontology.DataType*

classmethod base64(*length=0*)
Create a base64 DataType instance.

Parameters **length** (*int*) – Max number of bytes (zero = unlimited)

Returns

Return type *DataType*

classmethod enum(**choices*)
Create an enumeration DataType instance.

Parameters ***choices** (*str*) – Possible string values

Returns

Return type *edxml.ontology.DataType*

classmethod uri(*path_separator='/'*)
Create an URI DataType instance.

Parameters **path_separator** (*str*) – URI path separator

Returns

Return type *edxml.ontology.DataType*

classmethod hex(*length*, *separator=None*, *group_size=None*)
Create a hexadecimal number DataType instance.

Parameters

- **length** (*int*) – Number of hex digits
- **separator** (*str*) – Separator character
- **group_size** (*int*) – Number of hex digits per group

Returns

Return type *edxml.ontology.DataType*

classmethod uuid()
Create a uuid DataType instance.

Returns

Return type *DataType*

classmethod geo_point()
Create a geographical location DataType instance.

Returns

Return type *edxml.ontology.DataType*

classmethod file()
Create a file DataType instance.

Returns

Return type *edxml.ontology.DataType*

classmethod ip_v4()
Create an IPv4 DataType instance

Returns

Return type *edxml.ontology.DataType*

classmethod ip_v6()
Create an IPv6 DataType instance

Returns

Return type *edxml.ontology.DataType*

get()
Returns the EDXML data-type attribute. Calling this method is equivalent to casting to a string.

Returns

Return type str

get_family()
Returns the data type family.

Returns

Return type str

get_split()
Returns the EDXML data type attribute, split on the colon (':'), yielding a list containing the individual parts of the data type.

Returns

Return type List[str]

is_numerical()
Returns True if the data type is of data type family 'number'. Returns False for all other data types.

Returns

Return type boolean

is_datetime()
Returns True if the data type is 'datetime'. Returns False for all other data types.

Returns

Return type boolean

is_valid_upgrade_of(*other*)
Checks if the data type is a valid upgrade of another data type.

Parameters **other** (DataType) – The other data type

Returns

Return type bool

normalize_objects(*values*)
Normalize values to valid EDXML object value strings

Converts each of the provided values into valid string representations for the data type. It takes an iterable as input and returns a set of normalized strings.

The object values must be appropriate for the data type. For example, numerical data types require values that can be cast into a number, string data types require values that can be cast to a string. Values of datetime data type may be datetime instances or any string that dateutil can parse. When inappropriate values are encountered, an EDXMLEventValidationError will be raised.

Parameters **values** (*Iterable[Any]*) – The input object values

Raises EDXMLEventValidationError

Returns Set[str]. The normalized object values

validate_object_value(*value*)
Validates the provided object value against the data type, raising an EDXMLValidationException when the value is invalid. The value may be a native type like an integer or a boolean. The value may also be a unicode encoded string, as it would appear in EDXML data.

Parameters **value** – Object value

Raises EDXMLEventValidationError

Returns

Return type *edxml.ontology.DataType*

validate()
Validates the data type definition, raising an EDXMLValidationException when the definition is not valid.

Raises `EDXMLOntologyValidationError`

Returns

Return type *edxml.ontology.DataType*

classmethod format_utc_datetime(*date_time*)
Formats specified dateTime object into a valid EDXML datetime string.

Notes

The datetime object must have its time zone set to UTC.

Parameters **date_time** (*datetime.datetime*) – datetime object

Returns EDXML datetime string

Return type str

EventSource

class edxml.ontology.**EventSource**(*ontology*, *uri*, *description='no description available'*, *acquisition_date=None*)
Bases: `edxml.ontology.ontology_element.VersionedOntologyElement`

Class representing an EDXML event source

get_uri()
Returns the source URI

Returns

Return type str

get_description()
Returns the source description

Returns

Return type str

get_acquisition_date()
Returns the acquisition date as a datetime object or None in case no acquisition date is set.

Returns The date

Return type Optional[datetime.datetime]

get_acquisition_date_string()
Returns the acquisition date as a string of None in case not acquisition date is set.

Returns The date in yyyymmdd format

Return type Optional[str]

get_version()
Returns the version of the source definition.

Returns

Return type int

set_description(*description*)
Sets the source description

Parameters **description** (*str*) – Description

Returns The EventSource instance

Return type *edxml.ontology.EventSource*

set_acquisition_date(*date_time*)
Sets the acquisition date

Parameters **date_time** (*datetime.datetime*) – Acquisition date

Returns The EventSource instance

Return type *edxml.ontology.EventSource*

set_acquisition_date_string(*date_time*)
Sets the acquisition date from a string value

Parameters **date_time** (*str*) – The date in yyyymmdd format

Returns The EventSource instance

Return type *edxml.ontology.EventSource*

set_version(*version*)
Sets the concept version

Parameters **version** (*int*) – Version

Returns The Concept instance

Return type *edxml.ontology.Concept*

validate()
Checks if the event source definition is valid.

Raises EDXMLOntologyValidationError

Returns The EventSource instance

Return type *edxml.ontology.EventSource*

update(*source*)
Updates the event source to match the EventSource instance passed to this method, returning the updated instance.

Parameters **source** (edxml.ontology.EventSource) – The new EventSource instance

Returns The updated EventSource instance

Return type *edxml.ontology.EventSource*

generate_xml()
Generates an lxml etree Element representing the EDXML <source> tag for this event source.

Returns The element

Return type etree.Element

OntologyElement

class edxml.ontology.**OntologyElement**
Bases: abc.ABC

Class representing an EDXML ontology element

VersionedOntologyElement

class edxml.ontology.**VersionedOntologyElement**
Bases: edxml.ontology.ontology_element.OntologyElement

An ontology element that is versioned, such as an object type, concept, event type or an event source.

get_version()
Returns the version of the ontology element

Returns Element version

Return type int

describe_producer_rst()

edxml.ontology.description.**describe_producer_rst**(*ontology*, *producer_name*, *input_description*)
Returns a reStructuredText description for a producer of an ontology, such as a transcoder or a processor.

Parameters

- **ontology** (edxml.ontology.Ontology) – The ontology
- **producer_name** (*str*) – Name of the ontology producer
- **input_description** (*str*) – Short description of the data used as input by producer

Returns reStructuredText description

Return type str

generate_graph_property_concepts()

edxml.ontology.visualization.**generate_graph_property_concepts**(*ontology*, *graph*)
Appends nodes and edges to specified Digraph that show possible concept mining reasoning paths.

Parameters

- **ontology** (edxml.ontology.Ontology) –
- **graph** (*graphviz.Digraph*) –

edxml.ontology.visualization.**parent_child_hierarchy**(*ontology*, *graph*)
Appends nodes and edges to specified Digraph that show parent-child relations between all event types in the ontology.

Parameters

- **ontology** (edxml.ontology.Ontology) –
- **graph** (*graphviz.Digraph*) –

3.7 EDXML Ontology Bricks

EDXML ontology bricks are commonly used to share object types and concepts. The EDXML Foundation keeps an online repository of shared definitions.

Bricks can be registered with the *Ontology* class. After registering a brick, you can create an event type and refer to the object types and concepts from the brick. The ontology will automatically fetch the referred ontology elements from the registered brick and include them in the ontology. The following example illustrates this:

```
from edxml.ontology import Brick, Ontology


# Define an ontology brick
class MyBrick(Brick):
    @classmethod
    def generate_object_types(cls, target_ontology):
        yield target_ontology.create_object_type('my.object.type')


# Register brick with Ontology class
Ontology.register_brick(MyBrick)

# Now we can refer to the object type in the brick
ontology = Ontology()
event_type = ontology.create_event_type('my.event.type')
event_type.create_property('prop', object_type_name='my.object.type')
```

3.7.1 edxml.ontology.Brick

class edxml.ontology.**Brick**

Bases: object

Class representing an ontology brick. Ontology bricks contain definitions of object types and concepts. By defining these in ontology bricks, the definitions can be shared and installed using standard Python package management tools.

Using ontology bricks simplifies the process of producing and maintaining collections of tools that generate mutually compatible EDXML data streams, by sharing ontology element definitions in the form of Python modules.

Ontology bricks should extend this class and override the generate methods that create the ontology elements.

classmethod generate_object_types(*target_ontology*)

Creates any object types that are defined by the brick using specified ontology, yielding each of the created ObjectType instances.

Parameters **target_ontology** (edxml.ontology.Ontology) – The ontology to add to

Yields *List[edxml.ontology.ObjectType]*

classmethod generate_concepts(*target_ontology*)

Creates any concepts that are defined by the brick using specified ontology, yielding each of the created Concept instances.

Parameters **target_ontology** (edxml.ontology.Ontology) – The ontology to add to

Yields *List[edxml.ontology.Concept]*

classmethod test()

This method can be used in unit tests to verify ontology bricks.

3.8 Event Type Factories

Apart from procedurally defining ontology elements it is also possible to define them in a declarative fashion. This is done by defining a class that extends *EventTypeFactory*. This class defines a set of class constants that can be populated to describe one or more event types. These event types can then be generated from these class constants.

A quick example to illustrate:

```
from edxml.ontology import EventTypeFactory


class MyFactory(EventTypeFactory):
    TYPES = ['event-type.a']
    TYPE_PROPERTIES = {
        'event-type.a': {'property-a': 'my.object.type'}
    }


ontology = MyFactory().generate_ontology()
```

The EventTypeFactory class is also the base of the *RecordTranscoder* class and its extensions. These classes combine ontology generation with event generation and are the most convenient way to generate EDXML data.

Of course there are many more class constants corresponding to many other EDXML ontology features. These can be found in the below class documentation. More extensive examples of using an EventTypeFactory can be found in *Data Transcoding* and in *Introduction to EDXML Data Modelling*.

3.8.1 edxml.ontology.EventTypeFactory

class edxml.ontology.**EventTypeFactory**

Bases: object

This class allows for the creation of event types in a declarative fashion. To this end it provides a collection of class constants from which one or more event types can be generated.

VERSION = 1

The VERSION attribute indicates the version of the event types that are generated by this record transcoder. It can be increased when the event types are changed. This allows merging of EDXML data that was generated by both older and newer versions of the record transcoder.

TYPES = []

The TYPES attribute contains the list of names of event types that will be generated by the factory.

TYPE_DESCRIPTIONS = {}

The TYPE_DESCRIPTIONS attribute is a dictionary mapping EDXML event type names to event type descriptions.

TYPE_DISPLAY_NAMES = {}

The TYPE_DISPLAY_NAMES attribute is a dictionary mapping EDXML event type names to event type display names. Each display name is a list, containing the singular form, optionally followed by the plural form, like this:

```
{'event-type-name': ['event', 'events']}
```

The plural form may be omitted. This can be done by omitting the second item in the list or by using a string in stead of a list. In that case, the plural form will be assumed to be the singular form with an additional 's' appended.

TYPE_SUMMARIES = {}
The TYPE_SUMMARIES attribute is a dictionary mapping EDXML event type names to event summary templates.

TYPE_STORIES = {}
The TYPE_STORIES attribute is a dictionary mapping EDXML event type names to event story templates.

TYPE_PROPERTIES = {}
The TYPE_PROPERTIES attribute is a dictionary mapping EDXML event type names to their properties. For each key (the event type name) there should be a dictionary containing the desired property. A property dictionary must have a key containing the property name and a value containing the name of the object type. Properties will automatically be generated when this constant is populated.

Example:

```
{'event-type-name': {'property-name': 'object-type-name'}}
```

TYPE_PROPERTY_CONCEPTS = {}
The TYPE_PROPERTY_CONCEPTS attribute is a dictionary mapping EDXML event type names to property concept associations. The associations are dictionaries mapping property names to their associated concepts. The associated concepts are specified as a dictionary containing the names of the associated concepts as keys and their association confidences as values.

Example:

```
{
  'event-type-name': {
    'property-a': {'concept.name': 8},
    'property-b': {'concept.name': 8}, {'another.concept.name': 7},
  }
}
```

TYPE_PROPERTY_CONCEPTS_CNP = {}
The TYPE_PROPERTY_CONCEPTS_CNP attribute is a dictionary mapping EDXML event type names to property concept naming priorities (CNP). The priorities are specified as dictionaries mapping property names to concept CNPs. The concept CNPs are specified as a dictionary containing the names of the associated concepts as keys and their CNPs as values. When the CNP of a concept association is not specified, it will have the default value of 128.

Example:

```
{
  'event-type-name': {
    'property-a': {'concept.name': 192},
    'property-b': {'concept.name': 64}, {'another.concept.name': 0},
  }
}
```

TYPE_PROPERTY_ATTRIBUTES = {}
The TYPE_PROPERTY_ATTRIBUTES attribute is a dictionary mapping EDXML event type names to concept attributes. The concept attributes are specified as a dictionary mapping property names to attributes. Each attribute is a list containing the full attribute name, the singular display name and the plural display name, in that order. When the plural display name is omitted, it will be guessed by taking the singular form and appending an 's' to it. When both singular and plural display names are omitted, they will be inherited from the object type. When the attribute of a concept association is not specified, it will inherit from the object type as per the EDXML specification.

Example:

```
{
  'event-type-name': {
    'property-a': {'concept.name': ['object.type.name:attribute.name-extension
↪']},
    'property-b': {'concept.name': ['object.type.name:attribute.name-extension
↪', 'singular', 'plural']},
  }
}
```

TYPE_UNIVERSALS_NAMES = {}

The TYPE_UNIVERSALS_NAMES attribute is a dictionary mapping EDXML event type names to name relations. The name relations are dictionaries mapping properties containing object values (source properties) to other properties containing names for those object values (target properties). Name relations are discussed in detail in the EDXML specification.

Note that the source properties are automatically marked as being single-valued.

Example:

```
{
  'event-type-name': {
    'product-id': 'product-name'
  }
}
```

TYPE_UNIVERSALS_DESCRIPTIONS = {}

The TYPE_UNIVERSALS_DESCRIPTIONS attribute is a dictionary mapping EDXML event type names to description relations. The description relations are dictionaries mapping properties containing object values (source properties) to other properties containing description for those object values (target properties). Description relations are discussed in detail in the EDXML specification.

Note that the source properties are automatically marked as being single-valued.

Example:

```
{
  'event-type-name': {
    'product-id': 'product-description'
  }
}
```

TYPE_UNIVERSALS_CONTAINERS = {}

The TYPE_UNIVERSALS_CONTAINERS attribute is a dictionary mapping EDXML event type names to container relations. The container relations are dictionaries mapping properties containing object values (source properties) to other properties providing 'containers' for those object values (target properties). Container relations are discussed in detail in the EDXML specification.

Note that the source properties are automatically marked as being single-valued.

Example:

```
{
  'event-type-name': {
    'product-name': 'product-category'
  }
}
```

TYPE_PROPERTY_DESCRIPTIONS = {}

The TYPE_PROPERTY_DESCRIPTIONS attribute is a dictionary mapping EDXML event type names to

property descriptions. Each property description is a dictionary mapping property names to descriptions. It will be used to automatically set the descriptions of any automatically generated properties.

Example:

```
{
  'event-type-name': {
    'property-a': 'description',
    'property-b': 'description'
  }
}
```

TYPE_PROPERTY_SIMILARITY = {}

The TYPE_PROPERTY_SIMILARITY attribute is a dictionary mapping EDXML event type names to property similarities. Each property similarity is a dictionary mapping property names to EDXML 'similar' attributes. It will be used to automatically set the similar attributes of any automatically generated properties.

Example:

```
{
  'event-type-name': {
    'property-a': 'similar attribute',
    'property-b': 'similar attribute'
  }
}
```

TYPE_PROPERTY_MERGE_STRATEGIES = {}

The TYPE_PROPERTY_MERGE_STRATEGIES attribute is a dictionary mapping EDXML event type names to property merge strategies. Each property merge strategy is a dictionary mapping property names to EDXML 'merge' attributes, which indicate the merge strategy of the property. It will be used to set the merge attribute for any automatically generated properties.

When no merge strategy is given, automatically generated properties will have the default strategy, which is 'match' for hashed properties and 'any' for all other properties.

For convenience, the EventProperty class defines some class attributes representing the available merge strategies.

Example:

```
{
  'event-type-name': {
    'property-name': EventProperty.MERGE_ADD
  }
}
```

TYPE_HASHED_PROPERTIES = {}

The TYPE_HASHED_PROPERTIES attribute is a dictionary mapping EDXML event type names to lists of hashed properties. The lists will be used to set the 'match' merge strategy for the listed properties. Example:

```
{'event-type-name': ['hashed-property-a', 'hashed-property-b']}
```

PARENTS_CHILDREN = []

The PARENTS_CHILDREN attribute is a list containing parent-child event type relations. Each relation is a list containing the event type name of the parent, the EDXML 'parent-description' attribute and the event type name of the child event type, in that order. It will be used in conjunction with the CHIL-

DREN_SIBLINGS and PARENT_MAPPINGS attributes to configure event type parents for any automatically generated event types. Example:

```
PARENTS_CHILDREN = [
  ['parent-event-type-name', 'containing', 'child-event-type-name']
]
```

Note: Please refer to the EDXML specification for details about how to choose a proper value for the parent-description attribute.

CHILDREN_SIBLINGS = []

The CHILDREN_SIBLINGS attribute is a list containing child-siblings event type relations. Each relation is a list containing the event type name of the child, the EDXML 'siblings-description' attribute and the event type name of the parent event type, in that order. Example:

```
CHILDREN_SIBLINGS = [
  ['child-event-type-name', 'contained in', 'parent-event-type-name']
]
```

Note: Please refer to the EDXML specification for details about how to choose a proper value for the siblings-description attribute.

PARENT_MAPPINGS = {}

The PARENT_MAPPINGS attribute is a dictionary mapping EDXML event type names to parent property mappings. Each mapping is a dictionary containing properties of the event type as keys and properties of the parent event type as values. Example:

```
PARENT_MAPPINGS = {
  'child-event-type-name': {
    'child-property-name-a': 'parent-property-name-a',
    'child-property-name-b': 'parent-property-name-b',
  }
}
```

Note: Please refer to the EDXML specification for details about how parent property mappings work.

TYPE_TIME_SPANS = {}

The TYPE_TIME_SPANS attribute contains the names of the properties that define the start and the end of the time spans of the events. When no property is set that defines the start or end of the event time spans, the start or end is implicitly defined to be the smallest or largest datetime value in the event, regardless of the property that contains the datetime value. By setting specific properties for the start and / or end of the time span, only the datetime values of that property define the start or end of the event timespan.

The attribute is a dictionary mapping EDXML event type names to tuples that define the time span properties. The first value in each tuple is the name of the property that defines the start of the event time spans, the second defines the end. By default, both are unset. Example:

```
{'event-type-name': ['timespan-start', 'timespan-end']}
```

TYPE_VERSIONS = {}

The TYPE_VERSIONS attribute contains the names of the properties that define the versions of the events.

The attribute is a dictionary mapping EDXML event type names to the names of the event properties that contain the version. Example:

```
{'event-type-name': 'version'}
```

TYPE_SEQUENCES = {}

The TYPE_SEQUENCES attribute contains the names of the properties that define the sequence numbers of the events. Together with event time spans, sequence numbers determine logical event order.

The attribute is a dictionary mapping EDXML event type names to the names of the event properties that contain the sequence number. Example:

```
{'event-type-name': 'sequence'}
```

TYPE_ATTACHMENTS = {}

The TYPE_ATTACHMENTS attribute is a dictionary mapping EDXML event type names to attachment names. Example:

```
{'event-type-name': ['my-attachment', 'another-attachment']}
```

TYPE_MULTI_VALUED_PROPERTIES = {}

The TYPE_MULTI_VALUED_PROPERTIES attribute is a dictionary mapping EDXML event type names to lists of property names that will be multi-valued. Example:

```
{'event-type-name': ['my-property', 'another-property']}
```

TYPE_OPTIONAL_PROPERTIES = {}

The TYPE_OPTIONAL_PROPERTIES attribute is a dictionary mapping EDXML event type names to lists of property names that will be optional. Example:

```
{'event-type-name': ['my-property', 'another-property']}
```

It is also possible to indicate that all event properties of a particular event type will be optional, like this:

```
{'event-type-name': True}
```

In this case TYPE_MANDATORY_PROPERTIES can be used to specify exceptions.

TYPE_MANDATORY_PROPERTIES = {}

All event properties are created as mandatory properties by default, unless TYPE_OPTIONAL_PROPERTIES indicates otherwise. When TYPE_OPTIONAL_PROPERTIES indicates that all event properties of a specific event type must be optional, TYPE_MANDATORY_PROPERTIES can list exceptions. It is a dictionary mapping EDXML event type names to lists of property names that will be mandatory. Example:

```
{'event-type-name': ['my-property', 'another-property']}
```

TYPE_ATTACHMENT_MEDIA_TYPES = {}

The TYPE_ATTACHMENT_MEDIA_TYPES attribute is a dictionary mapping EDXML event type names to attachment media types. The attachment media types are a dictionary mapping attachment names to its RFC 6838 media type. Example:

```
{'event-type-name': {'my-attachment': 'text/plain'}}
```

TYPE_ATTACHMENT_DISPLAY_NAMES = {}

The TYPE_ATTACHMENT_DISPLAY_NAMES attribute is a dictionary mapping EDXML event type names to attachment display names. The attachment display names are a dictionary mapping attachment

names to display names. Each display name is a list, containing the singular form, optionally followed by the plural form, like this:

```
{'event-type-name': {'my-attachment': ['attachment', 'attachments']}}
```

The plural form may be omitted. This can be done by omitting the second item in the list or by using a string in stead of a list. In that case, the plural form will be assumed to be the singular form with an additional 's' appended.

TYPE_ATTACHMENT_DESCRIPTIONS = {}

The TYPE_ATTACHMENT_DESCRIPTIONS attribute is a dictionary mapping EDXML event type names to attachment descriptions. The attachment descriptions are a dictionary mapping attachment names to its description. Example:

```
{'event-type-name': {'my-attachment': 'Just some attachment'}}
```

TYPE_ATTACHMENT_ENCODINGS = {}

The TYPE_ATTACHMENT_ENCODINGS attribute is a dictionary mapping EDXML event type names to attachment encodings. The attachment encodings are a dictionary mapping attachment names their encodings. Valid encodings are either 'unicode' or 'base64' Example:

```
{'event-type-name': {'my-attachment': 'unicode'}}
```

generate_ontology(*target_ontology=None*)

Generates the ontology elements, adding them to a target ontology. The target ontology is returned. By default, the target ontology is an empty ontology which is not validated after populating it. When another target ontology is passed, it will be updated using the generated ontology elements. This update operation does trigger ontology validation.

Parameters **target_ontology** (edxml.ontology.Ontology) –

Returns The resulting ontology

Return type *edxml.ontology.Ontology*

create_concepts(*ontology*)

This method may be used to define EDXML concepts that are referred to by the generated event types.

create_object_types(*ontology*)

This method may be used to define EDXML object types that are referred to by the generated event types.

create_event_types(*ontology*)

This method generates all event types using the class constants of this event type factory. The event types are created in specified ontology. This ontology must have all required object types and concepts.

classmethod create_event_type(*event_type_name*, *ontology*)

Creates specified event type in the provided ontology using the class attributes of this factory. It returns the resulting event type. The resulting event types can optionally be tuned by overriding this method.

Parameters

- **event_type_name** (*str*) – Name of the desired event type
- **ontology** (edxml.ontology.Ontology) – Ontology to add the event type to

Returns edxml.ontology.EventType

3.9 Concept Mining

The SDK contains a concept mining implementation that builds an in-memory graph from event data. The subject of EDXML concept mining is briefly touched in Introducing EDXML and explained more extensively here.

3.9.1 The Knowledge Base

A *knowledge base* stores two types of information originating from EDXML data:

- Concept instances
- Universals

As detailed in the EDXML specification, universals are things like names and descriptions for object values, originating from specific property relations.

A knowledge base can be populated using a *Miner*. By feeding EDXML events and ontologies to a Miner it will incrementally extract universals and store them inside its knowledge base. Feeding EDXML data will also grow the internal *reasoning graph* of the Miner. When all EDXML data is loaded, the reasoning graph is complete and concept mining can be triggered using the *mine()* method.

Rather than feeding *EDXMLEvent* and *Ontology* objects to a Miner it is also possible to parse EDXML data directly into a Miner. That can be done using either a *KnowledgePullParser* or a *KnowledgePushParser*.

A quick example to illustrate:

```
import os

from edxml.miner.knowledge import KnowledgeBase
from edxml.miner.parser import KnowledgePullParser


# Parse some EDXML data into a knowledge base.
kb = KnowledgeBase()
parser = KnowledgePullParser(kb)
parser.parse(os.path.dirname(__file__) + '/input.edxml')

# Now mine concept instances using automatic seed selection.
parser.miner.mine()

# See how many concept instances were discovered
num_concepts = len(kb.concept_collection.concepts)
```

Concept Mining Seeds

Concept mining always needs a starting seed. A starting seed is a specific event object that is used as a starting point for traversing the reasoning graph. The mining process will then 'grow' the concept by iteratively adding adjacent event objects in the graph to the concept. Just calling the `mine()` method without any arguments will automatically find suitable seeds and mine them until all event objects in the graph have been assigned to a concept instance. Instead of automatic seed selection, a seed can be passed to the `mine()` method. That will cause only this one seed to be mined and a single concept being added to the knowledge base.

Class Documentation

The class documentation can be found below.

- *Miner*
- *KnowledgeBase*
- *KnowledgePullParser*
- *KnowledgePushParser*
- *KnowledgeParserBase*

Miner

class edxml.miner.**Miner**(*knowledge_base*)
Bases: object

Class combining an ontology, concept graph and a knowledge base to mine concepts and universals.

Parameters **knowledge_base** (edxml.miner.knowledge.KnowledgeBase) – Knowledge base to use

mine(*seed=None*, *min_confidence=0.1*, *max_depth=10*)
Mines the events for concept instances. When a seed is specified, only the concept instance containing the specified seed is mined. When no seed is specified, an optimum set of seeds will be selected and mined, covering the full event data set. The algorithm will auto-select the strongest concept identifiers. Any previously obtained concept mining results will be discarded in the process.

After mining completes, the concept collection is updated to contain the mined concept instances.

Concept instances are constructed within specified confidence and recursion depth limits.

Parameters

- **seed** (EventObjectNode) – Concept seed
- **min_confidence** (*float*) – Confidence cutoff
- **max_depth** (*int*) – Max recursion depth

KnowledgeBase

class edxml.miner.knowledge.**KnowledgeBase**
Bases: object

Class that can be used to extract knowledge from EDXML events. It can do that both by mining concepts and by gathering universals from name relations, description relations, and so on.

concept_collection = None
The concept instance collection holding mined concept instances.

get_names_for(*object_type_name*, *value*)
Returns a dictionary containing any names for specified object type and value. The dictionary has the object type names of the names as keys. The values are sets of object values.

Parameters

- **object_type_name** (*str*) – Object type name
- **value** (*str*) – Object value

Returns Dict[str, Set]

get_descriptions_for(*object_type_name*, *value*)
Returns a dictionary containing any descriptions for specified object type and value. The dictionary has the object type names of the descriptions as keys. The values are sets of object values.

Parameters

- **object_type_name** (*str*) – Object type name
- **value** (*str*) – Object value

Returns Dict[str, Set]

get_containers_for(*object_type_name*, *value*)
Returns a dictionary containing any containers for specified object type and value. As described in the EDXML specification, containers are classes / categories that a value belongs to. The dictionary has the object type names of the containers as keys. The values are sets of object values.

Parameters

- **object_type_name** (*str*) – Object type name
- **value** (*str*) – Object value

Returns Dict[str, Set]

add_universal_name(*named_object_type*, *value*, *name_object_type*, *name*)
Adds a name universal. A name universal associates a value with a name for that value and is usually mined from EDXML name relations. The parameters are two pairs of object type / value combinations, one for the value that is being named and one for the name itself.

Parameters

- **named_object_type** (*str*) – Object type of named object
- **value** (*str*) – value of named object
- **name_object_type** (*str*) – Object type of name
- **name** (*str*) – Name value

add_universal_description(*described_object_type*, *value*, *description_object_type*, *description*)
Adds a description universal. A description universal associates a value with a description for that value and is usually mined from EDXML description relations. The parameters are two pairs of object type / value combinations, one for the value that is being described and one for the description itself.

Parameters

- **described_object_type** (*str*) – Object type of described object
- **value** (*str*) – value of described object
- **description_object_type** (*str*) – Object type of description
- **description** (*str*) – Description value

add_universal_container(*contained_object_type*, *value*, *container_object_type*, *container*)
Adds a container universal. A container universal associates a value with another value that contains it and is usually mined from EDXML container relations. The parameters are two pairs of object type / value combinations, one for the value that is being contained and one for the container itself.

Parameters

- **contained_object_type** (*str*) – Object type of contained object
- **value** (*str*) – value of contained object

- **container_object_type** (*str*) – Object type of container
- **container** (*str*) – Container value

filter_concept (*concept_name*)
Returns a copy of the knowledge base where the concept instances have been filtered down to those that may be an instance of the specified EDXML concept.

The universals are kept as a reference to the original knowledge base.

Parameters **concept_name** (*str*) – Name of the EDXML concept to filter on

Returns Filtered knowledge base

Return type *KnowledgeBase*

filter_attribute (*attribute_name*)
Returns a copy of the knowledge base where the concept instances have been filtered down to those that have at least one value for the specified EDXML attribute.

The universals are kept as a reference to the original knowledge base.

Parameters **attribute_name** (*str*) – Name of the EDXML concept attribute to filter on

Returns Filtered knowledge base

Return type *KnowledgeBase*

filter_related_concepts (*concept_ids*)
Returns a copy of the knowledge base where the concept instances have been filtered down to those that are related to any of the specified concept instances.

The universals are kept as a reference to the original knowledge base.

Parameters **concept_ids** (*Iterable[str]*) – Iterable containing concept IDs

Returns Filtered knowledge base

Return type *KnowledgeBase*

to_json (*as_string=True*, ***kwargs*)
Returns a JSON representation of the knowledge base. Note that this is a basic representation which does not include details such as the nodes associated with a particular concept attribute.

Optionally a dictionary can be returned in stead of a JSON string.

Parameters

- **as_string** (*bool*) – Returns a JSON string or not
- ****kwargs** – Keyword arguments for the json.dumps() method.

Returns JSON string or dictionary

Return type Union[dict, str]

classmethod from_json (*json_data*)
Builds a KnowledgeMiner from a JSON string that was previously created using the to_json() method of a concept instance collection.

Parameters **json_data** (*str*) – JSON string

Returns

Return type *KnowledgeBase*

KnowledgePullParser

class edxml.miner.parser.**KnowledgePullParser**(*knowledge_base*)
Bases: *edxml.miner.parser.KnowledgeParserBase*, edxml.parser.EDXMLPullParser

EDXML pull parser that feeds EDXML data into a knowledge base.

Parameters **knowledge_base** (KnowledgeBase) – Knowledge base to use

KnowledgePushParser

class edxml.miner.parser.**KnowledgePushParser**(*knowledge_base*)
Bases: *edxml.miner.parser.KnowledgeParserBase*, edxml.parser.EDXMLPushParser

EDXML push parser that feeds EDXML data into a knowledge base.

Parameters **knowledge_base** (KnowledgeBase) – Knowledge base to use

KnowledgeParserBase

class edxml.miner.parser.**KnowledgeParserBase**(*knowledge_base*)
Bases: object

Parameters **knowledge_base** (KnowledgeBase) – Knowledge base to use

miner = None
The Miner instance that is used to feed the EDXML data into

3.9.2 Concept Graph API

Concept graphs are collections of *Node* instances connected by edges. The edges represent inferences and are instances of the *Inference* class. Virtually all interaction with the graph takes place using the *ConceptInstanceGraph* class. While the nodes and edges are exposed when extracting mining results, they are not commonly accessed directly.

Mining is initiated using *seeds*. A seed is an initial graph node from which a concept instance is grown by iterative associative reasoning, taking the seed as starting point.

The graph can be mined in two different ways. Either the graph is mined from a specific seed selected by the API user. Alternatively, the API is requested to auto-select the most promising seed. Automatic seed selection and mining can be repeated to obtain a set of concept instances that 'covers' the entire graph, yielding a complete set of knowledge extracted from the event data.

Obtaining concept mining results is done by means of the *extract_result_set* method of the *ConceptInstanceGraph* class. It returns a MinedConceptInstanceCollection instance which is covered in detail *here*.

Class Documentation

The class documentation can be found below.

ConceptInstanceGraph

class edxml.miner.graph.**ConceptInstanceGraph**(*ontology=None*)
Bases: object

Class representing a graph of concept nodes. The graph can contain information about a single concept instance or about multiple concepts. Depending on the graph topology these instances may or may not be related.

add(*node*)

Parameters **node** (Node) –

mine(*seed=None*, *min_confidence=0.1*, *max_depth=10*)

Mines the graph for concept instances. When a seed is specified, only the concept instance containing the specified seed is mined. When no seed is specified, an optimum set of seeds will be selected and mined, covering the entire graph. The algorithm will auto-select the strongest concept identifiers spread across the graph as seeds. Any previously obtained concept mining results will be discarded in the process.

Concept instances are constructed within specified confidence and recursion depth limits.

Parameters

- **seed** (EventObjectNode) – Concept seed
- **min_confidence** (*float*) – Confidence cutoff
- **max_depth** (*int*) – Max recursion depth

find_optimal_seed(*max_taint=0*)

Finds and returns the optimal seed for constructing a new concept instance or None in case all nodes are badly tainted. The taint of a node is the confidence of the node being part of any previously mined concepts. A seed is considered optimal if it is a strong identifier of a concept.

Parameters **max_taint** (*float*) – Node taint limit

Returns

Return type Optional[*EventObjectNode*]

extract_result_set(*min_confidence=0.1*)

Extracts the concept mining results from the graph, skipping any results that have confidence below specified threshold.

Parameters **min_confidence** (*float*) – Confidence threshold

Returns

Return type *MinedConceptInstanceCollection*

reset()

Clears artifacts of previous concept mining from the graph

Node

class edxml.miner.**Node**(*object_type_name*, *value*, *confidence*)

Bases: object

object_type_name = None

The name of the object type associated with the node

value = None

The object value that is represented by the node

confidence = None

Confidence of the node

time_span = None

Time line of node confidence

reason = None
The reason of a node is a reference to one of the edges which was used during reasoning to arrive at this node.

conclusions = None
The conclusions of a node are references to zero or more of its edges which were used during reasoning to infer other nodes.

add_inward(*edge*)
Adds specified edge as an inward edge.

Parameters **edge** (Inference) –

add_outward(*edge*)
Adds specified edge as an outward edge.

Parameters **edge** (Inference) –

get_inferences()

Returns

Return type Iterable[*Inference*]

get_inter_concept_inferences()

Returns

Return type List[*Inference*]

get_intra_concept_inferences()

Returns

Return type List[*Inference*]

get_same_concept_inferences(*seed*, *min_confidence*)

Parameters

- **seed** (Node) – Concept seed
- **min_confidence** (*float*) – Minimum confidence

Returns

Return type List[*Inference*]

clear_edge_roles()
Clears the roles that the edges play as either a reason or an argument. These roles are specific to the perspective of a particular seed.

reset()
Resets the state of the node to its initial state, clearing the edge roles, marking the node as unvisited, and so on.

EventObjectNode

class edxml.miner.node.**EventObjectNode**(*event_id*, *concept_association*, *object_type_name*, *value*, *confidence*, *time_span*)
Bases: edxml.miner.node.Node

Node representing a single instance of an object value.

link_relation(*node*, *relation*)

Parameters

- **relation** (*edxml.ontology.PropertyRelation*) –
- **node** (EventObjectNode) –

Inference

class edxml.miner.inference.**Inference**(*source_node*, *target_node*, *confidence*)
Bases: object

An edge in a concept instance graph representing the inference of a relation between two nodes.

Parameters

- **source_node** (edxml.miner.Node) –
- **target_node** (edxml.miner.Node) –

reason(*seed*, *confidence*)
Performs a reasoning step by using this inference to go from the source node to the target node. When the target node was previously reasoned to from any other source node, that source node will be detached from the target node first.

Parameters

- **seed** (edxml.miner.Node) – Seed of the concept instance
- **confidence** – New confidence of target node

compute_dijkstra_confidence(*seed*)
Returns the confidence of the target node for use as edge length when using Dijkstra's algorithm for finding the shortest path to a given node.

Parameters **seed** (edxml.miner.Node) – Concept seed

Returns

Return type float

3.9.3 Concept Mining Results

Concept mining results are represented by the *ConceptInstanceCollection* class. These are basically just collections of *ConceptInstance* objects. Concept instances expose lists of *ConceptAttribute* objects.

The concept attributes contain an object value, an EDXML object type name and one or more EDXML concept names that it is associated with. The names of the concept attributes consist of the name of an object type, a colon and optionally an extension, as per the EDXML specification .

The *from_json()* function can be used to re-create a concept instance collection from a previously generated JSON string.

Class Documentation

The class documentation of the various result classes can be found below.

- *ConceptInstanceCollection*
- *ConceptInstance*
- *ConceptAttribute*

- *MinedConceptInstanceCollection*
- *MinedConceptInstance*

ConceptInstanceCollection

class edxml.miner.result.**ConceptInstanceCollection**(*concepts=None*)
Bases: object

A collection of concept instances.

concepts = None
Dictionary of concept instances. Keys are unique concept identifiers.

to_json(*as_string=True*, ***kwargs*)
Returns a JSON representation of the concept instance collection. Note that this is a basic representation which does not include details such as the nodes associated with a particular concept attribute.

Optionally a dictionary can be returned in stead of a JSON string.

Parameters

- **as_string** (*bool*) – Returns a JSON string or not
- ****kwargs** – Keyword arguments for the json.dumps() method.

Returns JSON string or dictionary

Return type Union[dict, str]

ConceptInstance

class edxml.miner.result.**ConceptInstance**(*identifier*)
Bases: object

attributes = None
List of concept attributes.

id
An opaque identifier of the concept instance within the collection that it is part of.

Returns

Return type Any

add_attribute(*attribute*)
Adds an attribute to the instance.

Parameters **attribute** (ConceptAttribute) – Attribute

add_related_concept(*concept_id*, *confidence*)
Adds another related concept instance from the same concepts collection.

Parameters

- **concept_id** (*str*) – Concept identifier
- **confidence** (*float*) – Relation confidence

get_concept_names()
Compiles the names of all possible concepts that this concept may be an instance of. Returns a dictionary

containing the concept names as keys and their confidences as values. Confidences are given as a floating point number in range [0,1]

Returns

Return type dict

get_best_concept_name()
Returns the name of the most likely EDXML concept that this is an instance of.

Returns

Return type str

get_instance_title()
Finds and returns the attribute value that is most suitable for use as a title for the concept instance.

Returns

Return type str

get_related_concepts()
Get information about other concept instances from the same collection that may be related. Returns a dictionary containing the identifiers of related concept instances as keys and the confidence of the relation as values.

Returns Related concepts

Return type Dict[str, float]

get_attributes(*name*)
Returns the list of all attributes that have specified name.

Parameters **name** (*str*) – Attribute name

Returns

Return type List[*ConceptAttribute*]

ConceptAttribute

class edxml.miner.result.**ConceptAttribute**(*name*, *value*, *confidence=1.0*, *confidence_timeline=()*, *concept_naming_priority=128*, *concept_names=None*)

Bases: object

The ConceptAttribute class represents a single attribute of a concept instance, viewed from the perspective of a specific seed. It holds the collection of inferred nodes that confirm the existence of the attribute.

Parameters

- **name** (*str*) – Attribute name
- **value** (*str*) – Attribute value
- **confidence** (*float*) – Attribute Confidence
- **confidence_timeline** (*List[Tuple[datetime.datetime,datetime.datetime,float]]*) – Time line of confidences
- **concept_naming_priority** (*int*) – Concept naming priority
- **concept_names** (*Dict[str, float]*) – Concept names and confidences

Returns

Return type *ConceptAttribute*

object_type_name
The name of the EDXML object type of the attribute.

Returns

Return type str

confidence
The confidence is the likelihood that the attribute belongs to the concept.

Returns Confidence

Return type float

confidence_timeline
The confidence timeline shows how the likelihood that the attribute belongs to the concept changes over time.

Returns Confidence timeline

Return type List[List[datetime.datetime,datetime.datetime, float]]

concept_naming_priority
Returns the concept naming priority of the attribute, which determines how suitable the attribute is for naming a concept instance.

Returns

Return type int

concept_names
Returns a dictionary containing the names of all concepts that refer to this attribute as keys and their confidences as values.

Returns Dict[str, float]

edxml.miner.result.**from_json**(*json_data*)
Builds a ConceptInstanceCollection from a JSON string that was previously created using the to_json() method of a concept instance collection.

Parameters **json_data** (*str*) – JSON string

Returns

Return type *ConceptInstanceCollection*

The following two classes are extensions exposing graph details like nodes and inferences.

MinedConceptInstanceCollection

class edxml.miner.result.**MinedConceptInstanceCollection**
Bases: *edxml.miner.result.ConceptInstanceCollection*

get_seeds()
Get the seeds from all concepts in the result set

Returns

Return type List[*Node*]

MinedConceptInstance

class edxml.miner.result.**MinedConceptInstance**(*seed_id*)
Bases: *edxml.miner.result.ConceptInstance*

Class representing a single mined concept instance and its attributes.

get_nodes()
Returns a NodeCollection containing all nodes which are part of the concept.

Returns

Return type NodeCollection

get_seed()
Returns the seed node

Returns

Return type *Node*

get_related_concepts()
Get information about other concept instances from the same collection that may be related. Returns a dictionary containing the identifiers of related concept instances as keys and the confidence of the relation as values.

Returns Related concepts

Return type Dict[str, float]

3.9.4 Concept Graph Visualization

The SDK contains some functions for using GraphViz to visualize the concept graph.

edxml.miner.graph.visualize

edxml.miner.graph.visualize.**graphviz_nodes**(*concepts*, *graph=None*)
Adds concept instances to a GraphViz directed graph instance and returns it. By default a suitable GraphViz instance is generated, using an instance created by the caller is also possible.

The resulting graph is very detailed, showing every single node. As such, it is intended to be used for small graphs.

Parameters

- **concepts** (edxml.miner.result.MinedConceptInstanceCollection) – Concept instances
- **graph** (*graphviz.Digraph*) – GraphViz Digraph instance

Returns GraphViz Digraph instance

Return type graphviz.Digraph

edxml.miner.graph.visualize.**graphviz_concepts**(*concepts*, *graph=None*)
Adds concept instances to a GraphViz directed graph instance and returns it. By default a suitable GraphViz instance is generated, using an instance created by the caller is also possible.

The resulting graph shows the concepts, their attributes and the reasoning paths that connect the attributes within the concept. Attributes that are shared among multiple concept instances are generally not connected,

which keeps the graphs simple and readable. An exception is made for ambiguities where it is unclear which concept instance an attribute belongs to. A heuristic is used to decide what to do.

Parameters

- **concepts** (edxml.miner.result.MinedConceptInstanceCollection) – Concept instances
- **graph** (*graphviz.Digraph*) – GraphViz Digraph instance

Returns GraphViz Digraph instance

Return type graphviz.Digraph

CHAPTER 4

Command Line Utilities

An overview of included command line utilities can be found here.

CHAPTER 5

Indices and tables

- genindex
- search

Python Module Index

e

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

X