EDXML

Specification Document

edxml.org
version 3.0.0

PUBLISHED BY THE EDXML FOUNDATION

www.edxml.org

Acknowledgements

A special thanks goes to

The Dutch National Police Agency *and* Northwave Nederland B.V.

for providing the challenges that inspired this effort,
for their role as incubators

Pim Takkenberg *and* Tycho van Marle

for their championing of this effort
from the very beginning

Contents

1. Introduction

This document is the official specification of the EDXML data representation. EDXML offers a single generic representation for virtually any structured data, allowing different kinds of data from various sources to be forged into a single consistent data set. At the same time EDXML is also a simple form of knowledge representation. As such, it enables machines to learn the meaning of data, learn how to automatically correlate various types of data, reason about it and assist human analysts in connecting the dots to discover the big picture.

EDXML originates from operational needs and was shaped into its current form by applying it to actual real world problems in forensics, law enforcement, intelligence and cyber security. As such, it has a strong focus on simplicity and practical applicability.

Detailed information about the origin, background and ideas that drive the EDXML effort can be found on www.edxml.org.

1.1 Valid EDXML

An EDXML document is valid if and only if both of the following two conditions are met:

1. The EDXML document validates against the EDXML RelaxNG schema[1]
2. The EDXML document adheres to the specifications outlined in this document.

[1] This schema can be obtained from https://github.com/edxml/schema/

1.2 General EDXML structure

EDXML documents use the <edxml> tag as root element and are encoded using the UTF-8 character encoding. Unless indicated otherwise, all XML elements in EDXML documents must be in the EDXML namespace. The EDXML namespace URI is http://edxml.org/edxml.

EDXML represents all data by means of *events*. While the name 'event' suggests that it represents an occurrence taking place at some point in time, its representational scope is much wider. An EDXML event is probably best described as follows:

An environment providing coherence and
context for a group of one or more data elements.

In this specification document we will frequently use phone call records from an imaginary Client Relations Management (CRM) system of a

web shop as examples of events. A phone call event may contain a set of data elements which includes the phone numbers involved, a time stamp and a duration. These data elements are called *value objects*, or objects for short. Each event groups several objects together and provides the context for these objects. Due to the context (phone call), the objects are given meaning, and their mutual relations are defined. Events may contain other kinds of data elements besides objects, such as attachments.

The general layout of an EDXML document is shown in figure 1. Every EDXML document consists of two different interleaving elements: Elements describing ontology information (green) and elements describing the events (blue). These two elements may repeat themselves indefinitely to form documents of arbitrary size.

An ontology element contains all of the information that is required to interpret and validate the events that follow it.

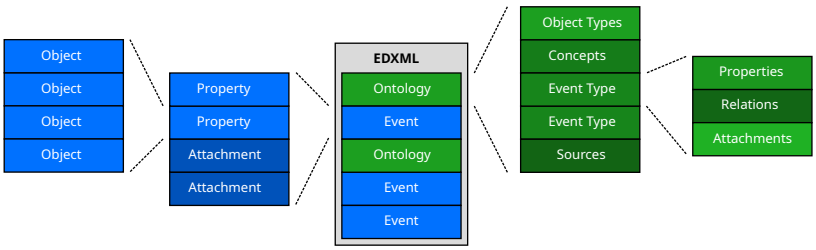

Figure 1: General structure of an EDXML document. Note that this is not a complete representation of all elements of an EDXML document.

1.3 Events and Ontologies

Events are represented by means of `<event>` elements which have the root `<edxml>` element as their parent. Any EDXML document that contains at least one event must also contain at least one `<ontology>` element. Like the `<event>` elements, `<ontology>` elements must be children of the root element. The first ontology element must precede the first event element. There is no limit to the number of `<event>` and `<ontology>` elements that an EDXML document can contain. A coarse[1] example of this layout is shown below.

[1] For brevity we occasionally show partial XML structures by replacing sub-elements or attributes with ellipsis (…) and specifying the content of these parts elsewhere.

```
<edxml version="3.0.0" xmlns="http://edxml.org/edxml">
  <ontology>
  ...
  </ontology>
  <event event-type=... source-uri=...>
  ...
  </event>
  <ontology>
  ...
  </ontology>
  <event event-type=... source-uri=...>
  ...
  </event>
  ...
</edxml>
```

The `version` attribute of the root element indicates the version of the EDXML specification that is to be applied when interpreting the document. The EDXML specification uses semantic versioning[1].

[1] Please refer to https://semver.org/.

1.3.1 Event Elements

Each event must have an `event-type` and a `source-uri` attribute. The `event-type` attribute contains the name of one of the event types[2] defined in any of the preceding `<ontology>` elements. Any event must be valid according to the requirements imposed by its event type. The `source-uri` attribute must contain the URI of one of the event sources[3] defined in any of the preceding `<ontology>` elements.

[2] Event types are specified in section 2

[3] Event sources are specified in section 3

An example of an event is shown below.

```
<event event-type=... source-uri=...>
  <properties>
    <caller>0034656286219</caller>
    <callee>0034642772906</callee>
    <duration>5</duration>
  </properties>
</event>
```

The `<properties>` element can be populated with zero or more objects. An object is a child of the properties element having a tag that matches the name of any of the properties defined by the event type. Note that this implies that property names must be valid XML tag names. The use of XML tag names which are reserved by the W3C XML specification is not allowed. The value of the object is the text content of the object element. Generally, an event may contain zero or more objects for each of the properties of the event type. Event types may restrict the number of event objects that a property can contain[4].

[4] Refer to section 2.2 for details

Object values must not be empty. An empty object value in EDXML is represented by omitting the object entirely.

Events may have one or more attachments. While properties play a key role in defining the structure of events, attachments do not. While object values are typically short strings, attachments are intended to store longer strings, like JSON documents for example. We can extend the above phone call event to contain a note about the conversation, like this:

```
<event event-type=... source-uri=...>
  <properties>
    <caller>0034656286219</caller>
    <callee>0034642772906</callee>
    <duration>5</duration>
  </properties>
  <attachments>
    <note id="note">Client requested 10% discount.</note>
  <attachments/>
</event>
```

The `<attachments>` element is optional. When present it must be populated with one or more attachments. An attachment is a subelement of the `<attachments>` element having a tag that matches the name of any of the attachments defined by the event type. The value of the attachment is the text content of the attachment element. Each attachment element must have an `id` attribute which uniquely identifies an attachment amongst all attachments in the event that share a common attachment name. Within a single event there must not be more than one attachment having a particular combination of attachment name and attachment identifier.

Attachments must not be empty. An empty attachment is represented by omitting it entirely.

An `<event>` element may have an optional `parents` attribute containing a comma separated list of parent events. The attribute is intended to indicate that an event has been created using one or more other events (the 'parents') as input. It aids in tracking an event back to its origins.

The parent events must be specified by means of their sticky hashes[1] as computed using the SHA1 hashing function and represented as lowercase hexadecimal strings of 40 characters in length.

[1] Sticky hashes are specified in section 8.

1.3.2 Ontology Elements

An `<ontology>` element may contain any number of ontology components. The various types of ontology components are:

- Event type
- Object type
- Concept
- Source

Ontology elements must define at least the ontology components that are referred to by the events between it and either the next ontology element or the end of the document. Events must not refer to an ontology component unless that component has been previously defined in the same EDXML document.

The general structure of an ontology element looks like this:

```
<ontology>
  <object-types>
    <object-type>
    ...
    </object-type/>
  </object-types>
  <concepts>
    <concept>
    ...
    </concept/>
  </concepts>
  <event-types>
    <event-type>
    ...
    </event-type>
  </event-types>
  <sources>
    <source>
    ...
    </source>
  </sources>
</ontology>
```

2. Event Types

Event types form the very fabric of EDXML documents. They provide the context that defines the role of each of the objects in an event, their value spaces, how they are related and what their use is for data analysis. This section specifies how to define them.

Contents

2.1 Defining an event type

An event type is defined by adding an `<event-type>` element as a child of an `<ontology>` element. An abbreviated example of a phone call event type definition, including a number of event properties, is given below:

```
<event-type name="acme.crm.phonecall"
            display-name-singular="phone call"
            display-name-plural="phone calls"
            description="sales team phone call record"
            version="1"
            summary=... story=... >
  <property name="caller"
            description="caller"
            optional="false"
            multivalued="false"
            confidence="10"
            object-type="telecom.phone.number"/>
  <property name="callee"
            description="callee"
            optional="false"
            multivalued="false"
            confidence="10"
            object-type="telecom.phone.number"/>
  <property name="duration"
            description="duration"
            optional="false"
            multivalued="false"
            confidence="10"
            object-type="datetime.duration.seconds"/>
</event-type>
```

The various attributes of the `<event-type>` element are specified below.

2.1.1 The name attribute

The `name` attribute uniquely identifies the event type. It must be composed of one or more components, separated by dots. The dotted structure is used to place event type names in a virtual namespace hierarchy. Namespacing is intended to reduce the risk of name collisions when merging EDXML documents produced by various parties. In the above example, we used the name `acme.crm.phonecall` to identify phone calls stored in a CRM system from vendor Acme Corporation. The name is constructed from left to right, starting with the name of the vendor of the CRM software. Each subsequent name component makes the name more specific.

2.1.2 The description attribute

The description attribute must contain a clear and concise description of the event type.

2.1.3 The display name attributes

The `display-name-singular` and `display-name-plural` attributes contain the singular and plural forms of the display name of the event type. The display name provides a human friendly version of the event type name intended for display in user interfaces, visualizations, reports and so on.

2.1.4 The story and summary attributes

The `story` and `summary` attributes are EDXML templates[1] that can be used to construct a textual representation of the information in an event, in human readable form. The `story` template is intended to generate a full description of the event. The `summary` attribute is intended to generate a short description that can be used in contexts where space is limited. It should not refer to more than one or two properties.

[1] EDXML templates are specified in section 11.

2.1.5 The event-version attribute

The `event-version` attribute is used for versioning of events. When set, the event type becomes a versioned event type. The attribute has two uses:

- Detect event merge conflicts[2]
- Merging of properties that use the `replace` merge strategy[3]

[2] Event merge conflicts are defined in section 2.2.8
[3] Event merge strategies are defined in section 2.2.8

The `event-version` attribute is optional except when the event type defines any properties that use the `replace` merge strategy. The `event-version` attribute must contain the name of a mandatory single-valued property of the event type. This property must have data type `sequence`[4] and must have the `max` merge strategy.

[4] Data types are defined in section 5.9

2.1.6 The sequence attribute

The `sequence` attribute is used to determine the ordering of events. The attribute is fully specified in section 2.7.1

2.1.7 The timespan attributes

The `timespan-start` and `timespan-end` attributes are used to determine how dates and times in the events must be interpreted. Both attributes are fully specified in section 2.6.

2.2 Adding properties

Let us return to our initial example of an event type definition, which defines some properties:

```
<event-type name="acme.crm.phonecall"
            display-name-singular="phone call"
            display-name-plural="phone calls"
            description="sales team phone call record"
            version="1"
            summary=... story=... >
  <property name="caller"
            description="caller"
            optional="false"
            multivalued="false"
            confidence="10"
            object-type="telecom.phone.number"/>
  <property name="callee"
            description="callee"
            optional="false"
            multivalued="false"
            confidence="10"
            object-type="telecom.phone.number"/>
  <property name="duration"
            description="duration"
            optional="false"
            multivalued="false"
            confidence="10"
            object-type="datetime.duration.seconds"/>
</event-type>
```

The various attributes of the `<property>` element are specified below.

2.2.1 The name attribute

The property name uniquely identifies the property within the scope of its event type. As such, the event type acts as a name space for properties. Different event types may define properties with identical names.

Property names may contain dots (.). The use of dots in property names is intended to create an implicit hierarchical structure in the properties of an event type. As an example, consider the following set of event property names:

```
customer.name
customer.address.city
customer.address.street
```

This structure can be useful when translating EDXML events into a JSON representation for example. The dotted structure may be used to create JSON objects that look like this:

```
{
  'customer': {
    'name': 'Alice',
    'address': {
      'city': 'London',
      'street': 'Oxford Street'
    }
  }
}
```

2.2.2 The description attribute

The description attribute must contain a concise description of the property which clarifies the role of the property within the context of the event type. When the property name is sufficiently descriptive of its own, the description may be identical to the property name.

2.2.3 The object-type attribute

The `object-type` attribute must contain the name of any of the object types defined in the same `ontology` element or a preceding one in the same EDXML document. The object type determines the value space of the objects of the property. Events must not contain objects for this property other than those within the value space of the object type. Object types are specified in section 5.

2.2.4 The optional attribute

When the `optional` attribute is set to `true`, the property may be omitted in events. We will refer to these properties as optional properties. An attribute value of `false` makes the property mandatory, which means that events must have at least one object for the property. [1]

[1] We recommend making event properties mandatory when possible, as it provides a safe assumption that data analysis algorithms can make about the data.

2.2.5 The multivalued attribute

When the `multivalued` attribute is set to `true`, the property may have multiple objects in a single event. We will refer to these properties as multi-valued properties. An attribute value of `false` makes the property single-valued: It must not have more than one object for any single event.[2]

[2] We recommend making event properties single-valued when possible, as it provides a safe assumption that data analysis algorithms can make about the data.

2.2.6 The confidence attribute

The `confidence` attribute is an integer number in the interval [1, 10] which indicates the probability that object values of the property are correct.

Attribute values smaller than 10 can be used for properties of which the object values may be inaccurate. For example, an event type representing customer data that originates from manual input that has not been verified may be given a lower confidence value.

2.2.7 The similar attribute

The optional `similar` attribute provides hints to aid in finding events that are similar to another event of the same type. In the case of our phone call example, one might want to search the dataset for similar phone calls that originated from the same caller. To hint at this possibility the `similar` attribute may contain a short phrase that describes how one event relates to similar events that share the same property value.[1] The description should fit into a grammar construct like this:

[1] While an analyst could find similar events by matching the value of *any* property the intent of this attribute is to provide shortcuts to typical searches. These could be used to generate context menu items in a GUI for example.

Find all <plural event type display name> <similar attribute> *the same* <property description>

For instance, when a `similar` attribute value of *"originating from"* is used for property `caller` of event type `acme.crm.phonecall`, this might be used to generate the following query description:

Find all phone calls originating from the same caller

2.2.8 The merge attribute

An EDXML document may contain multiple physical events that represent instances of one and the same logical event. EDXML documents may be processed to merge these instances into one. The `merge` attributes of the event properties determine the *merge strategy* that must be used to merge the objects of each of the properties of the events. Event merge operations and the role of the `merge` attribute are fully specified in section 9.

The `merge` attribute is optional. When it is specified it must have one of the following values:

- `min`
- `max`
- `add`
- `replace`
- `match`
- `set`
- `any`

When the attribute is omitted its default value must be assumed to be 'any'.

Properties that use the `min` or `max` strategy must be mandatory and single-valued.

Properties that use the `replace` strategy must be optional and single-valued.

Properties that use the `min` or `max` strategy must have an object type that has a data type that is a member of the `number`, `sequence` or `datetime` family[1].

[1] Please refer to section 5.9 for details about data type families.

Properties that use the `match` strategy must not be associated with an object type which has data type `number:float` or `number:double`. This restriction originates from the fact that floating point values cannot always be converted between their decimal string representation and binary representation in a lossless way. Event hashes[2] must be guaranteed not to change when events are serialized to or deserialized from XML. Using floating point values in hashes would make it difficult to guarantee this.

[2] Event hashes are specified in section 8.

2.2.9 The version attribute

The `version` attribute determines the version of the event type definition. Versioning of ontology components is covered in section 6.

2.2.10 Concept associations

Properties may be associated with one or multiple concepts. These associations indicate that the property contains objects which are identifiers of the concept that it is associated with.[3] An example of a concept association is shown below.

[3] Please refer to section 10 to learn more about EDXML concepts and which role event properties play to define and construct concept instances.

```
<property name="caller"
          description="caller"
          optional="false"
          multivalued="false"
          confidence="10"
          object-type="telecom.phone.number">
          <property-concept name="person"
                            confidence="8"
                            cnp="128"/>
          </property>
```

In the above example a property named `caller` is associated with a concept named `person`. The concept in an association must be defined in the same `<ontology>` element or a preceding one in the same EDXML document. A property must not have multiple associations with the same concept.

The mandatory `confidence` attribute is used to express the *concept identi-*

fication authority of a property. Concept identification authority expresses how authoritative a property is for identifying a concept instance[1]. More precise:

> *The estimated probability*[2] *that the correct concept instance is selected from all possible concept instances in the dataset, when the only selection criterion is the object type and value of the property for which the authority is being defined.*

For example, consider a property that is an identifier for a concept representing a person. When that property contains passport numbers then it is a fairly strong identifier for that concept. Hence, it should be given a high concept confidence. On the other hand, in case that property contained family names then it should be given a low confidence for that same concept, because there could be many different persons in a dataset sharing the same family name.[3]

The `cnp` attribute is an integer in the interval [0,255] which contains the *Concept Naming Priority* (CNP) of the event property. The CNP is intended to be used by concept mining[4] applications that need to choose between different property objects to use as a name for a concept instance. Consider two properties that are both identifiers for the same concept. One property represents the name of a person and sets a high CNP. The other event property represents a passport number and sets a low CNP. Then, the name will be preferred over the passport number to name concept instances. This is especially useful to allow user interfaces to automatically pick the most suitable value to name concepts.

Each concept association implicitly defines a *concept attribute*. The role of concept attributes is similar to the role of properties in event types. Concept attributes are named after the object type of the associated property. The name of the attribute is defined to be the name of the object type followed by a colon (:) and an extension. By default, this extension is an empty string. The extension can be overridden by setting the XML attribute `attr-extension` of the `<property-concept>` tag to the desired extension.

Customizing the attribute name extension should only be done in cases where the object type is too generic to convey the meaning of the object values within the context of the concept. As an example consider an event type containing first names and family names of employees. Both of these are values of an object type named `person.name` and are associated with a concept named `person`. This implies that instances of the `person` concept will have a single concept attribute named `person.name:` which will contain both first names and family names. This is the result of the context transfer that is taking place. When constructing concept instances, object values are 'transferred' from event properties to concept attributes. While the names originate from an event type that stores first names and family names in separate properties, they share the same object type. As a result,

[1] Refer to section 10 for information about concepts and concept instances

[2] On an integer scale in the interval [0,10]

[3] Given that there are only 11 possible confidence values, determining the best suitable value should not be taken to be an exact science. A rough estimate will do.

[4] Refer to section 10 for information about concept mining.

first names and family names will populate the same concept attribute by default. By setting the `attr-extension` attributes of the associations in this example to `first` and `family` the `person.name:` attribute is split into two distinct concept attributes, `person.name:first` and `person.name:family`.

By default concept attributes inherit their display names from the object types of the associated property. When an attribute name extension is specified, the display names of the attribute can be overridden as well by setting custom display names in the `attr-display-name-singular` and `attr-display-name-plural` attributes of the `<property-concept>` tag.

2.3 Adding property relations

Property relations can be added to an event type by adding a `<relations>` element. This element can have any number of child elements, each of which defines a property relation. Various types of property relations can be specified. The tag name of the property relation element indicates the relation type. Let us return to our initial example of an event type definition once again and add a property relation:

```
<event-type name="acme.crm.phonecall"
            display-name-singular="phone call"
            display-name-plural="phone calls"
            description="sales team phone call record"
            version="1"
            summary=... story=... >
  <property name="caller"
            ...
            object-type="telecom.phone.number">
            <property-concept cnp="128"
                              confidence="8"
                              name="person"/>
  </property>
  <property name="callee"
            ...
            object-type="telecom.phone.number">
            <property-concept cnp="128"
                              confidence="8"
                              name="person"/>
  </property>
  <property name="duration"
            ...
            object-type="datetime.duration.seconds"/>
  <relations>
    <inter source="caller"
           target="callee"
           source-concept="person"
           target-concept="person"
           description=...
           predicate="contacted"
           confidence="9"/>
  </relations>
</event-type>
```

This example contains one property relation definition, between the “caller” property and the “callee” property. The relation is of type *inter*. The vari-

ous types of relations are detailed in section 2.3.7. The property relation attributes are specified below.

2.3.1 The source and target attributes

The `source` and `target` attributes must contain the names of the properties of the event type that are involved in the relation. Which property should be chosen as the source and which as the target may or may not be arbitrary depending on the type of relation, as indicated in section 2.3.6.

2.3.2 The concept attributes

Relations of relation types `inter` and `intra` do not only relate two properties, they also relate the concepts associated with those properties. Relations of these types must specify both the `source-concept` and the `target-concept` attribute to indicate which concepts are involved in the relation. The `source-concept` attribute must be the name of one of the concepts associated with the source property. The `target-concept` attribute must be the name of one of the concepts associated with the target property.

2.3.3 The description attribute

The `description` attribute contains an EDXML template. The general format of EDXML templates is specified in section 11. In addition to these general specifications, the description attribute must refer to both related properties. It must not refer to any of the other properties or attachments of the event type.

All types of relations must set the description attribute unless specified otherwise in the sections that cover the relation types in more detail. The template describes the reason why the two properties are related, fitting into a grammar construct like this:

The properties are related because <relation description>

For example, the description attribute of the defined relation between the caller and callee in our phone call example might read:

```
a phone call was registered between [[caller]] and [[callee]]
```

2.3.4 The predicate attribute

The `predicate` attribute provides a concise alternative to the value of the `description` attribute. All types of relations must set this attribute unless specified otherwise in the sections that cover the relation types in more detail. As in English grammar the predicate is intended to be placed in between a subject and an object where, in this case, the subject is a value of the source property and the object is a value of the target property. For example, the following predicates

- is married to
- communicates with
- is called

could be processed into the following relation descriptions:

- Alice *is married to* Bob
- Alice *communicates with* Bob
- Alice *is called* Miss Piggy

2.3.5 The confidence attribute

The `confidence` attribute is defined as follows:

The probability[1] *that the relation actually exists when it is observed once*

[1] scaled to an integer number in the interval [0, 10]

Here, an observation of a relation is defined as an event having an object for both related properties. When any of the two properties has no object in a particular event then the relation does not exist within the context of that event. Also, when an event has multiple object values for any of the related properties, then all possible combinations of object values of the two properties are observations of the relation. For example, when an event has n objects for property `source` and m objects for property `target`, then a total of $n \cdot m$ observed relations result.

All types of relations must set this attribute unless specified otherwise in the sections that cover the relation types in more detail.

2.3.6 Relation Types

The EDXML specification defines the following relation types:

inter	This type of relation interconnects one concept instance[1] to another. For example, it may relate a particular customer to a particular product.
intra	This type of relation relates information about the same concept instance. It may, for instance, relate the name of a person to the home address of that same person.
name	Identifies one property as providing names for the objects of another property.
description	Identifies one property as providing descriptions for the objects of another property.
container	Identifies the object of one property as being part of the object of another property.
original	Identifies the object of one property as being the original of the object of another property.
other	Generic type used for relations for which none of the other relation types are suitable.

[1] The meaning of *concepts* and *concept instances* is detailed in section 10

An event type must not define more than one relation for each possible combination of relation type, `source`, `target`, `source-concept` and `target-concept`.

Note that a relation does not relate a pair of properties or concepts *in general*. Depending on the presence or absence of objects in a given event a relation may or may not exist in that specific event.

The various types of relations are specified in detail in the following sections.

2.3.7 intra

An intra-concept relation is intended to convey that the objects of both related properties are attributes of a single concept instance. Machines can

use these relations to discover new information about a given concept instance. Which property is the source and which is the target is an arbitrary choice.

The two concepts in an intra-concept relation are commonly chosen to be identical. The concepts may also be chosen to be different. There are two possible interpretations of intra-concept relations between two different concepts:

specialization One concept in the relation may be a specialization[1] of the other. A specialized concept is defined as an extension of the concept name created by appending to it. For example, when `source-concept` has value `person` while `target-concept` has value `person.customer` then `target-concept` is a specialization of `source-concept`. The interpretation of this intra-concept relation is discovery of the fact that a given `person` instance is actually a `person.customer`.

combination When the two related concepts differ and one is not a specialization of the other then the intra-concept relation is to be interpreted as discovery that a given concept instance is also an instance of the related concept. The resulting concept instance is a combination of both concepts.

[1] In linguistics this is commonly called a *hyponym*.

2.3.8 inter

An inter-concept relation is intended to convey that the objects of both related properties are attributes of two distinct concept instances. Both related concepts (`source-concept` and `target-concept`) may be either identical or they may differ. Which property is the source and which is the target is an arbitrary choice.

2.3.9 name

Name relations relate two properties where the source property provides names for the values of the target property. For example, an event type could define two properties containing a product number and a product name. A name relation can be used to explicitly specify this relationship.

Name relations are intended to generate *universals*, which means that if the relation between two objects exists within the context of one event then it exists in any context[2]. More precisely, when a given event contains a name relation which states that object $\mathcal{O}_s$ of object type $\mathcal{T}_s$ is a name for object

[2] Due to this it is not recommended to define name relations involving object types that are highly generic in nature.

$\mathcal{O}_t$ of object type $\mathcal{T}_t$ then $\mathcal{O}_s$ of object type $\mathcal{T}_s$ may *always* be used as an appropriate name for object $\mathcal{O}_t$ of object type $\mathcal{T}_t$, no matter which event the object appears in.

The source property of a name relation must be single-valued. Name relations must not set a `description` attribute, `confidence` attribute or `predicate` attribute.

2.3.10 description

Description relations relate two properties where the source property provides descriptions for the objects of the target property. For example, an event type could define two properties containing a product number and a product description. A description relation can be used to explicitly specify this relationship.

The only difference between name relations and description relations is that computers may assume that objects of the source property of a description relation are long compared to a name relation. Applications presenting objects to users may prefer to use a name in stead of a description when available presentation space is limited.

Description relations are intended to generate *universals*, which means that if the relation between two objects exists within the context of one event then it exists in any context[1]. More precisely, when a given event contains a description relation which states that object $\mathcal{O}_s$ of object type $\mathcal{T}_s$ is a description for object $\mathcal{O}_t$ of object type $\mathcal{T}_t$ then $\mathcal{O}_s$ of object type $\mathcal{T}_s$ may *always* be used as an appropriate description for object $\mathcal{O}_t$ of object type $\mathcal{T}_t$, no matter which event the object appears in.

[1] Due to this it is not recommended to define description relations involving object types that are highly generic in nature.

The source property of a description relation must be single-valued. Description relations must not set a `description` attribute, `confidence` attribute or `predicate` attribute.

2.3.11 container

In a container relation the objects of the source property conceptually contain the objects of the target property. For example, an event type might define two properties containing product categories and product names. A container relation can be used to explicitly specify that the category contains the product.

Container relations are intended to generate *universals*, which means that if the relation between two objects exists within the context of one event then it exists in any context[2]. More precisely, when a given event contains a container relation which states that object $\mathcal{O}_s$ of object type $\mathcal{T}_s$ is a container for object $\mathcal{O}_t$ of object type $\mathcal{T}_t$ then $\mathcal{O}_s$ of object type $\mathcal{T}_s$ may *always* be regarded as containing object $\mathcal{O}_t$ of object type $\mathcal{T}_t$, no matter which event the object appears in.

[2] Due to this fact it is not recommended to define container relations involving object types that are highly generic in nature.

The source property of a container relation must be single-valued. Container relations must not set a `description` attribute, `confidence` attribute or `predicate` attribute.

2.3.12 original

In an original relation the object of the source property contains the original value of the object of the target property. The intended use case is the situation where the original values from a data source need to be normalized in order to be represented using a particular EDXML object type. For example, a mixed case string value that is not case sensitive may be represented using a lowercase string to facilitate correlating the values. Another example is converting the time zone of a date / time value in order to obtain a valid EDXML `datetime` object, which is always represented as UTC. In cases like these keeping both the normalized value and the original value may be desirable. An original relation can be used to explicitly specify that some property contains the original value of another property.

Both the source property and the target property of an original relation must be single-valued. Original relations must not set a `description` attribute, `confidence` attribute or `predicate` attribute.

2.3.13 other

A relation of type `other` is a general purpose relation type that can be used to define relations for which no other suitable relation type exists. Which property is the source and which is the target is an arbitrary choice.

2.4 Adding a parent definition

An event type can optionally identify another event type as its parent. This means that the events of the child have a many-to-one relationship with an event of the parent event type. Adding a `<parent>` element to the event type definition allows for specifying this relationship. This element must be the first child element of a `<event-type>` element. Any event type can have at most one parent event type. Consider the following example:

```
<event-type name="acme.crm.phonecall" ... >
  <parent event-type="acme.crm.client"
          property-map="client-id:cid"
          parent-description="associated with"
          siblings-description="related to"/>
  ...
</event-type>
```

In the above example, we express the fact that every phone call event is related to an event describing the details of the client that was involved in the call. We do that by defining a parent for the `acme.crm.phonecall` event type, in this example an event of type `acme.crm.client`. Next, we link the two event types by means of properties containing the client identifier, as is expressed in the `property-map` attribute. In our example both event types have a property containing the client identifier. So, given any phone call event, we can now find its associated client record by finding the event of type '`acme.crm.client`' of which the value of the `cid` property matches the value of the `client-id` property in the phone call event. Likewise, we can find the siblings of any given phone call event by finding all phone call events that have the same parent event.

The various attributes of the `<parent>` element are specified below.

2.4.1 The event-type attribute

The `event-type` attribute specifies the name of the parent event type. The parent event type must be defined in the same `ontology` element or a preceding one in the same EDXML document.

2.4.2 The property-map attribute

The `property-map` attribute maps child properties to matching properties in the parent. The `property-map` attribute must not be empty and it must provide a mapping to each of the hashed properties[1] of the parent.[2]

A property mapping is specified as a string containing the name of a child property followed by a colon and the name of a property of the parent. Multiple property mappings are specified by concatenating property mappings with a comma between the mappings. An example of a property mapping string that maps multiple properties is given below:

```
prop1:prop1,prop2:prop2
```

Note that finding the parent of an event requires matching all objects of the properties contained in the property map as well as the object count. For example, the child event might not have any objects for a property contained in the property map. In that case, the parent event must also have no objects for the corresponding property.

Each child property in the `property-map` must have a merge strategy of either `match` or `any`. These restrictions assure that events have one fixed parent event which cannot be changed by merging it with another event.

[1] Hashed properties are defined in section 9.1

[2] This implies that every child event has exactly one parent.

2.4.3 The parent-description attribute

The `parent-description` attribute contains a string describing the parent in relation to the child. The string must fit in a grammar construct like this:

... the <parent event type display name> <*parent-description*> this <child event type display name>

For example, when the parent event type has display name "client record", the child event type has display name "phone call" and `parent-description` is "associated with" then these strings may be combined to yield the following text phrase:

the client record associated with this phone call

2.4.4 The siblings-description attribute

The `siblings-description` attribute contains a string describing the siblings in relation to the parent. The string must fit in a grammar construct like this:

... all <child event type display name> <*siblings-description*> the same <parent event type display name>

For example, when the parent event type has display name "client record", the child event type has plural display name "phone calls" and `siblings-description` is "related to" then these strings may be combined to yield the following text phrase:

all phone calls related to the same client record

2.5 Defining attachments

Event types may contain an `<attachments>` element to define attachments that can be contained in its events. This element contains zero or more `<attachment>` elements to define each of the attachments.

An example of an attachment definition is shown below:

```
<attachment name="avatar"
            display-name-singular="user avatar"
            display-name-plural="user avatars"
            description="avatar of user [[user]]"
            media-type="image/png"
            encoding="base64"/>
```

As an example, the `<attachments>` element of an EDXML event having an event type containing the above attachment definition may look like this:

```
<attachments>
  <avatar id="default">YW55IGNhcm5hbCBwbGVhcw==</avatar>
</attachments>
```

The attributes of the `<attachment>` element are specified below.

2.5.1 The name attribute

The `name` attribute uniquely identifies the attachment within the event type.

2.5.2 The display name attributes

The `display-name-singular` and `display-name-plural` attributes contain the singular and plural forms of the display name of the attachment. The display name provides a human friendly version of the attachment name intended for display in user interfaces, visualizations, reports and so on.

2.5.3 The description attribute

The `description` attribute must be a valid EDXML template[1] that may contain references to the properties of the event type, allowing the attachment to be associated with zero or more event properties. The intent of the description is to describe how the attachment relates to the event.

[1] EDXML templates are specified in section 11

2.5.4 The media-type attribute

The `media-type` attribute must be a valid RFC 6838 media type which indicates how the attachment content should be interpreted.

2.5.5 The encoding attribute

The `encoding` attribute defines how the attachment content in the events is encoded. Its value must be one of the following:

base64	The attachment is base64 encoded. This encoding must be used when the attachments may contain characters that cannot be encoded as UTF-8 or characters that are not legal in XML documents.
unicode	The attachment is UTF-8 encoded. This encoding should be used when base64 encoding is not required.

Base64 encoded attachments must be valid `base64Binary` values as defined in the W3C XML Schema language recommendation. The use of whitespace in base64 encoded attachments is allowed.

2.6 Time

Event types may or may not define properties that represent time. Any property that refers to data type `datetime`[1] represents time. We will refer to the set of objects of these properties in a given event as the *time stamps* of that event. Depending on how an event type uses properties that represent time it can be either *timeless* or *timeful*. A timeless event type contains no properties that represent time and is typically used to represent facts that are time independent. A timeful event type contains at least one time representing property.

[1] See section 5 for details about object types and data types.

Any timeful event has a *time span*. The time span of a timeless event is undefined. Time spans are not explicitly specified for individual events. Rather, they are an interpretation of the time stamps of an event. The event time stamps translate into a time span by taking the smallest and largest of the time stamps.[2] In case an event contains just one time stamp, the start and end time stamps of its time span are both identical to the one time stamp in the event and the extent of the time span is zero. When an event of a timeful event type contains no time stamps, the time span of that event is the open interval $(-\infty, +\infty)$. Its extent is infinite.

[2] This is the case when the start or end of the time span is not explicitly defined by the event type, as specified below.

It is not always desirable to consider all time stamps contained in any event property to compute the time span. For example, consider an event type representing a ticket in an issue tracking system. During its life cycle, an issue might go through a number of different states and the time of each state change is recorded. This state tracking might be modelled by means of several time representing properties like `time-created`, `time-confirmed` and `time-resolved`. In this example, we may want the time span of tickets to represent the time between creation and reso-

lution. As long as the ticket is not resolved, no value is set for its `time-resolved` property. In this state, we would like its time span to have a beginning and no end. In other words, it has a half-open time interval.

Timeful event types that support closed, half-open and open time spans can be defined by means of two optional event type attributes `timespan-start` and `timespan-end`. Setting one or both of these attributes changes how the event time span is computed. When the `timespan-start` attribute is set to the name of one of the time representing properties of the event type, only the objects of that one property define the start of the time span. Now, the start of the time span is the lowest value of the objects of the specified property. In case that property has no values for a particular event, then the start of the time span is unknown, resulting in a (half-)open time interval. When the `timespan-end` attribute is set to the name of one of the time representing properties of the event type, only the objects of that property define the end of the time span. Now, the end of the time span is the highest value of the objects of the specified property. In case the property has no values for a particular event, the end of the time span is unknown, resulting in a (half-)open time interval. When both the `timespan-start` and `timespan-end` attributes are set and neither of these two properties has any objects for a particular event, the time span of that event is the open interval $(-\infty, +\infty)$ and its extent is infinite.

2.7 Order

The ordering of events as they appear in EDXML documents is insignificant. EDXML processing systems are not required to preserve the ordering of the events they receive as input. Still, knowledge about the ordering of the events may be crucial for some applications. For that reason we define the order of EDXML events in terms of the information contained in the events themselves.

When discussing event order we need to consider ordering of both logical and physical events. As detailed in section 2.2.8 multiple events in a given EDXML document may share the same sticky hash. This means that these physical events are instances of a single logical event. The ordering of logical events is relevant for cause-effect analysis. The ordering of physical events is relevant when merging multiple physical events in a single logical event, because merge operations[1] can be sensitive to ordering.

[1] Please refer to section 2.2.8 for information about event merge operations.

2.7.1 Logical event order

Determining the order for a set of logical events must be done by executing the following two sorting steps:

1. Sort on the start of the time spans of the events as defined in 2.6[1]
2. Sort any sets of events that have identical time span starting points on their *sequence numbers*

[1] Events having no time span start go first when ordered ascending.

The sequence numbers mentioned in step two are defined as follows. The `<event-type>` tag of an event type definition accepts an optional `sequence` attribute. This attribute must contain the name of a mandatory single-valued property of the event type. The object of this property is defined to be the sequence number of the event and must have the `sequence` data type[2]. Data sources can use the `sequence` attribute to provide more accurate ordering of its output events when time alone does not suffice.

[2] The sequence data type is defined in section 5.9.9

Step two of the logical event ordering procedure is not performed when the event type does not define a sequence number.

When two events have identical time span starting points and no sequence number is defined then the order of these events is undefined. When two events have identical time span starting points as well as identical sequence numbers then the order of these events is undefined.

2.7.2 Physical event order

The ordering of physical events is determined by means of the `event-version` attribute of the `<event-type>` tag of an event type definition. When merging physical events the values of this property must be used to sort the events. The *last* physical event in a set of events that represent the same logical event is defined as the event having the highest valued object of the `event-version` property.

Note that, while the ordering of events as they appear in EDXML documents is insignificant, there are two requirements for positioning events with respect to the definitions of their event types:

- Any event must have an `<ontology>` element preceding it which defines its event type, within the same EDXML document.
- Any event must be valid according to the newest[3] of all definitions of its event type which precede the event in the document.

[3] The newest definition is the one having the highest version in its `version` attribute.

2.8 Versioning and Upgrading

The same[1] event type may be defined multiple times, either within the same `<ontology>` element or in multiple `<ontology>` elements within the same EDXML document or across multiple documents. Each pair of these definitions within a single document must be either each others equivalents[2] or one must be a valid upgrade of the other. What constitutes a valid upgrade is defined in section 6.

[1] Here, *the same* means having identical values of their `name` attributes

[2] Equivalence is defined in section 12.2.4

In table 1 all attributes of the `event-type` tag are shown and if they can be upgraded or not.[3]

[3] Please refer to section 6 for details about ontology upgrading.

Table 1: Upgrading of event type attributes.

attribute	upgradable
`name`	n
`description`	y
`display-name-singular`	y
`display-name-plural`	y
`timespan-start`	n
`timespan-end`	n
`sequence`	n
`event-version`	n
`story`	y
`summary`	y

Upgrades must not remove event properties. Adding a property to an event type is permitted, provided that

- the new property is optional and
- adding the new property does not make a previously timeless event type into a timeful one.

The attributes of an existing property are partially upgradable as indicated in table 2.

Table 2: Upgrading of event property attributes.

attribute	upgradable
`name`	n
`description`	y
`object-type`	n
`merge`	n
`similar`	y
`confidence`	y
`optional`	y*
`multivalued`	y*

Some attributes are upgradable only when certain conditions are met. In ta-

ble 2 these are marked with a superscript asterisk (y*). For each of these attributes the conditions are detailed below.

optional — A mandatory event property can be upgraded to be optional but optional event properties cannot be upgraded to become mandatory.

multivalued — A single valued event property can be upgraded to be multi-valued but multi-valued event properties cannot be upgraded to become single-valued.

Upgrades must not remove property relations. Adding a property relation to an event type is permitted. The attributes of an existing property relation are partially upgradable as indicated in table table 3.

attribute	upgradable
`source`	n
`target`	n
`source-concept`	n
`target-concept`	n
`description`	y
`predicate`	y
`confidence`	y

Table 3: Upgrading of event property relation attributes.

Upgrades must not remove property / concept associations. Adding an association is permitted. The attributes of an existing property / concept association are partially upgradable as indicated in table 4.

attribute	upgradable
`name`	n
`attr-extension`	n
`attr-display-name-singular`	y
`attr-display-name-plural`	y
`confidence`	y
`cnp`	y

Table 4: Upgrading of property / concept association attributes.

Upgrades must not remove event type parent definitions. Adding a parent definition to an event type is permitted. The attributes of an existing parent definition are partially upgradable as indicated in table 5.

Upgrades must not remove event type attachment definitions. Adding an attachment definition to an event type is permitted. The attributes of an existing attachment definition are partially upgradable as indicated in table 6.

attribute	upgradable
event-type	n
property-map	n
parent-description	y
siblings-description	y

Table 5: Upgrading of event parent attributes.

attribute	upgradable
description	y
display-name-singular	y
display-name-plural	y
media-type	n
encoding	n

Table 6: Upgrading of event attachment attributes.

3. Event Sources

Any event refers to an event source by means of the event source URI in its source-uri attribute. Event sources are specified by adding <source> elements as children of the <sources> element of an ontology. A source definition looks like this:

```
<source uri="/some/source/uri/"
        description="client records from database X"
        date-acquired="20100128"
        version="1"/>
```

The same[1] source may be defined multiple times, either within the same <ontology> element or in multiple <ontology> elements within the same EDXML document or across multiple documents. Each pair of these definitions within a single document must be either each others equivalents[2] or one must be a valid upgrade[3] of the other.

The various attributes of the source element are specified below.

[1] Here, *the same* means having identical values of their uri attributes

[2] Equivalence is defined in section 12.2.3

[3] What constitutes a valid upgrade is defined in section 6

3.1 The uri attribute

The uri attribute uniquely identifies an EDXML event source. An event source indicates the origin of the events. An example of a source URI is given below:

```
/company/offices/germany/stuttgart/clientrecords/2009/
```

The collection of source URIs in an EDXML document may be conceived as a virtual tree, similar to a directory structure. Each URI uniquely identifies an EDXML data source within the tree. The URI must start and end with a slash. Note that the virtual directory structure is global, care should be taken to

choose a suitable prefix URI and use it to create a source URI namespace for the owner or producer of the data.

3.2 The description attribute

The `description` attribute can be used to provide more details about the event source.

3.3 The acquisition date attribute

The optional `date-acquired` attribute can be used to indicate how recent the information is. It is a six digit string in the format 'yyyymmdd', like '20100128'.

3.3.1 The version attribute

The `version` attribute determines the version of the event source definition. Details on versioning of ontology components are specified in section 6.

3.4 Versioning and Upgrading

The attributes of the `source` tag and their upgrading details are specified in table 7. Please refer to section 6 for details about ontology upgrading.

attribute	upgradable
`uri`	n
`description`	y
`date-acquired`	y

Table 7: Upgrading of event source attributes.

4. Concepts

EDXML concepts are a mechanism to relate properties from multiple event types that jointly describe the same thing. A concept might be a person, a location, a product or anything else that is relevant for a particular data set.[1] Concepts are defined by adding a `<concept>` child element to the `<concepts>` element of the ontology. Revisiting the CRM record example, we might want to define a client concept, which looks like this:

[1] Concepts are covered in more detail in section 10.

```
<concept name="person.client"
         display-name-singular="client"
         display-name-plural="clients"
         description="a customer of a particular vendor"
         version="1"/>
```

The same[2] concept may be defined multiple times, either within the same `<ontology>` element or in multiple `<ontology>` elements within the same EDXML document or across multiple documents. Each pair of these definitions within a single document must be either each others equivalents[3] or one must be a valid upgrade[4] of the other.

[2] Here, *the same* means having identical values of their `name` attributes

[3] Equivalence is defined in section 12.2.2

[4] What constitutes a valid upgrade is defined in section 6.

The various attributes of the `concept` element are described below.

4.1 The name attribute

The `name` attribute uniquely identifies the concept. It is composed of one or more components, separated by dots. The dotted structure is used to place concept names in a virtual hierarchy. Each of the components must be the hyponym of the component to the left (if any) and the hypernym of the component to the right (if any). In the above example, the name `person.client` consists of the `person` component and the `client` component where `client` is a hyponym of `person` while `person` is a hypernym of `client`. The concept `person.client` is called a *specialization* of the `person` concept while the `person` concept is called a *generalization* of the `person.client` concept.

Due to this hierarchical structure, EDXML documents which define concepts `a` and `a.b.c` should also define concept `a.b`, even when `a.b` is not referenced by any event type.

Concepts enable multiple data sources to contribute information about the same thing. However, this can only work when these data sources all refer to the same concept. For this reason concept definitions are often shared between data sources. Defining shared definitions is a joint effort of EDXML data source developers. Concepts that are not specific to a single data source should be publicly shared. The EDXML foundation maintains a public repository[5] for this purpose.

[5] See the Github project page at https://github.com/edxml/bricks

The EDXML RelaxNG schema allows concept names of up to 255 characters in length. This enables the use of deep concept hierarchies such as provided by

the Princeton WordNet lexicon.

Concept names carry implicit definitions of *universals*. In the field of knowledge representation universals are universal truths. In our example the `person.client` concept implies the following universals:

- a client is a kind of person
- a person may be a client

4.2 The display name attributes

The `display-name-singular` and `display-name-plural` attributes contain the singular and plural forms of the display name of the concept. The display name provides a human friendly version of the concept name intended for display in user interfaces, visualizations, reports and so on.

4.3 The description attribute

The `description` attribute should contain a clear and concise description of what exactly the concept represents.

4.4 The version attribute

The `version` attribute determines the version of the concept definition. Details on versioning of ontology components are specified in section 6.

4.5 Versioning and Upgrading

The attributes of the `concept` tag and their upgrading details are specified in table 8. Please refer to section 6 for details about ontology upgrading.

attribute	upgradable
`name`	n
`description`	y
`display-name-singular`	y
`display-name-plural`	y

Table 8: Upgrading of concept attributes.

5. Object Types

Every property in an event type definition must specify its object type by means of the `object-type` attribute. Object types serve two purposes. First, the object type determines the value space of valid values for the objects of the property. Event properties must not contain any objects outside the value space of its object type. Second, object types determine when two given objects are to be considered identical. This in turn enables correlating events that share common objects.

Two given objects, which may or may not be contained in the same EDXML document, are considered identical only when

1. the object types of both objects are either equivalents[1] of one another or one of the object types is a valid upgrade[2] of the other, *and*
2. the values of the objects are identical.

[1] Equivalence is defined in section 12.2.2

[2] What constitutes a valid upgrade is defined in section 6

Events produced by different data sources can have shared objects, which means that the events can be correlated. For this reason object type definitions are often shared between data sources. Defining shared definitions is a joint effort of EDXML data source developers. Object types that are not specific to a single data source should be shared. The EDXML foundation maintains a public repository[3] for this purpose.

[3] See the Github project page at https://github.com/edxml/bricks

An example of an object definition is shown below:

```
<object-type name="computing.networking.ipv4"
             display-name-singular="IPv4 address"
             display-name-plural="IPv4 addresses"
             description="internet IPv4 address in dotted
                 ↪ decimal notation"
             data-type="ip"
             compress="true"
             version="1"/>
```

The same[4] object type may be defined multiple times, either within the same `<ontology>` element or in multiple `<ontology>` elements within the same EDXML document or across multiple documents. Each pair of these definitions within a single document must be either each others equivalents[5] or one must be a valid upgrade[6] of the other.

[4] Here, *the same* means having identical values of their `name` attributes

[5] Equivalence is defined in section 12.2.1

[6] What constitutes a valid upgrade is defined in section 6.

Contents

The various attributes of the `<object-type>` element are specified next.

5.1 The name attribute

The `name` attribute uniquely identifies the object type. It must be composed of one or more components, separated by dots. The dotted structure is used to place object type names in a virtual namespace hierarchy. Namespacing is intended to reduce the risk of name collisions when merging EDXML documents produced by various parties. In the above example, we used the name `computing.networking.ipv4` to define an IPv4 address. The name is constructed from left to right, starting with the very generic name component 'computing'. Each subsequent name component makes the name more specific. The above example could be extended by introducing other object types that belong in the domain of computer networks, choosing names starting like `computing.networking`. Using this naming scheme is not required, but recommended. Object types that are specific to a particular product should have their own object type namespace. For example, an object type representing client identifiers in some CRM system from vendor 'Acme' might be named '`acme.crm.client-id`'

5.2 The description attribute

The `description` attribute should contain a clear and concise description of the values the object type is intended for.

5.3 The display name attributes

The `display-name-singular` and `display-name-plural` attributes contain the singular and plural forms of the display name of the object type. The display name provides a human friendly version of the object type name intended for display in user interfaces, visualizations, reports and so on.

5.4 The xref attribute

The optional `xref` attribute can be used to provide an URL to an external resource providing additional information about the object type. It might point to a Wikipedia article or a specification document for example.

5.5 The compress attribute

The optional `compress` attribute can be used to indicate that the object values are expected to have low entropy. Database systems that support compression can use this attribute to selectively compress objects of specific object types. When the attribute is not specified then its default value is assumed to be "false".

5.6 The regex attributes

The optional `regex-soft` and `regex-hard` attributes must contain a regular expression that is valid for use in a pattern facet as defined in the W3C XML Schema specification[1]. Both attributes apply only to object types having a data type from the `string` family. For object types having other data types, this attribute must not be set. If the `regex-hard` attribute is specified then properties that use the object type must not contain any objects which do not match the expression.

[1] Note that this implies that the expressions are implicitly anchored.

Contrary to `regex-hard` the `regex-soft` attribute must not be used to validate the objects in EDXML events. An EDXML event containing objects that do not match the `regex-soft` of their object type may still be a valid event. The intended use cases for the expression in `regex-soft` are

- to allow automatic identification of valid object values in external data sources, or
- to synthesize valid object values

When both `regex-hard` and a `regex-soft` are specified it is recommended that the `regex-soft` is either identical to `regex-hard` or that it is an expression that has a value space which is smaller than the value space of `regex-hard`.

5.7 The unit attributes

The optional `unit-name` and `unit-symbol` attributes can be used to indicate the name and symbol of the unit of measurement associated with numerical object types. The attributes must not be set for object types having data types other than those in the `number` family.

As an example, an object type representing a distance could specify "meters" as unit name and "m" as unit symbol. The International System of Units should be used whenever possible. The unit name must be specified in plural form. Metric prefixes like "kilo" / "k" or "milli" / "m" must not be used. The `unit-name` and `unit-symbol` attributes must both be specified or both omitted from object type definitions. Specifying one without the other is not allowed.

5.8 The prefix-radix attribute

The optional `prefix-radix` attribute can be used to indicate the natural base for metric prefixes of numerical data types. It is intended to be used in conjunction with the `unit-name` and `unit-symbol` attributes to display the values of objects. As such, it must not be set unless both the `unit-name` and `unit-symbol` attributes are set as well. The attribute can have any of the values 2, 10 and 60. The meaning of each is specified below:

2 Numeric IEC[1] prefixes should be used, which represent a multiple of 2. The "Ki" prefix represents 2^{10}, "Mi" represents 2^{20}, "Gi" represents 2^{30} and so on.

10 Numeric SI[2] prefixes should be used, which represent a multiple of 10. The "k" prefix represents 10^3, "M" represents 10^6, "G" represents 2^9 and so on.

60 Numeric prefixes represent a multiple of 60. The appropriate numerical prefixes are only defined when combined with specific unit symbols. When the unit symbol is "s" the values should be denoted as lengths of time. When the unit symbol is "°" (unicode character `U+00B0`) the values should be denoted as angles.

[1] IEC: International Electrotechnical Commission

[2] SI: Système international (d'unités)

If the `unit-name` and `unit-symbol` attributes are set while `prefix-radix` is not then the natural base must be assumed to be 10.

5.8.1 The version attribute

The `version` attribute determines the version of the object type definition. Details on versioning of ontology components are specified in section 6.

5.9 The data-type attribute

The `data-type` attribute determines the valid value space of the object type. The data types are subdivided into families. An overview of available data type families is displayed below.

number	Numerical data values (section 5.9.1)
hex	Values in hexadecimal notation (section 5.9.2)
geo	Geographical coordinates (section 5.9.3)
file	References to externally stored data files (section 5.9.5)
uri	RFC 3986 URIs (section 5.9.4)
uuid	Universally unique identifiers (section 5.9.6)
ip	Internet Protocol addresses (section 5.9.7)
datetime	ISO 8601 dates and times (section 5.9.8)
sequence	Sequential integer numbers (section 5.9.9)
boolean	Boolean values (section 5.9.10)
enum	String values from a predefined set (section 5.9.11)
base64	Base64 encoded bytes (section 5.9.12)
string	String values (section 5.9.13)

Data types are specified as strings which consist of one or more components separated by colons (:). The data type family must be the first component. In case the family is subdivided into members the member must be the second component. Depending on the data type family and member, the data type may require additional components to yield a valid data type. The data type families are specified in more detail in the following sections.

5.9.1 The number family

The *number* data type family consists of the following members:

tinyint	8-bit unsigned integer value
smallint	16-bit unsigned integer value
mediumint	24-bit unsigned integer value
int	32-bit unsigned integer value
bigint	64-bit unsigned integer value
float	single-precision floating point value

double	double-precision floating point value
decimal	fixed-point value
currency	fixed-point value representing an amount of money

All integer members[1] and the `decimal` member are unsigned by default. For each of these data types there is a signed variant which has 'signed' as the last component. To illustrate, the `number:bigint` data type is unsigned while the corresponding signed data type is `number:bigint:signed`.

[1] Specifically: `number:tinyint`, `number:smallint`, `number:mediumint`, `number:int` and `number:bigint`

For objects of the number data type family the following two restrictions apply:

- Objects must not have any leading '+' sign or leading zeros.
- Zero must be represented without a leading sign.

The above two restrictions do not apply to objects of the two floating point members.

The value space of the `number:float` data type is patterned after the lexical space of the `float` data type defined by the W3C XML Schema language recommendation, except that the three special values +INF, -INF and NaN are not included[2].

[2] The rationale is that support for representing and processing these values in programming languages and database products is generally limited, which leads to problems handling any data containing these values.

The value space of the `number:double` data type is patterned after the lexical space of the `double` data type defined by the W3C XML Schema language recommendation, except that the three special values +INF, -INF and NaN are not included.

The value spaces of both `number:float` and `number:double` include those values which require rounding in order to be encoded into IEEE 754 binary encoding. EDXML implementations may apply this rounding only to values that require it. Note that this may result in multiple coexisting physical instances of a single event while each instance contains slightly different floating point object values. This fact is to be taken into consideration when choosing merge strategies for a given event type. The `decimal` member could be a better choice in some cases.

The `decimal` member enables exact representation of decimal values as well as safe serialization and deserialization to and from EDXML. The desired number of digits must be specified by means of the third and fourth component in the data type definition, like this:

```
data-type="number:decimal:7:2"
```

In the above example the third component specifies the total number of digits (7) while the fourth component specifies the number of digits be-

hind the decimal point (2). So, in the example, the number can have at most five digits before and must have two digits after the decimal point. The length of the values of the decimal data type must match the specified number of digits after the decimal exactly, padding with zeros if necessary. The total number of decimals must be greater than zero. The total number of digits must not exceed 38[1]. The signed variant of a decimal number is created by appending `:signed`, as shown in the example below:

[1] This is the number of digits that most databases support and should fit the needs for virtually all practical purposes.

```
data-type="number:decimal:7:2:signed"
```

The `currency` member is used to represent amounts of money. It has the same value space as the `number:decimal:19:4:signed` data type. As such, its value space is in accordance with the Generally Accepted Accounting Principles (GAAP). Object types that use this data type should set their `unit-symbol` attribute to one of the alpha codes defined by ISO 4217.

5.9.2 The hex data types

The hex data type family can be used for representing byte sequences in hexadecimal notation, with an optional separator character. A hex data type must consist of at least two components, the second component being the length of the value in bytes. An example is shown below:

```
data-type="hex:6"
```

The above example describes a hexadecimal number of 6 bytes in length written as 12 hexadecimal digits. The hex data type may be extended by adding a digit group size component and a separator component, like this:

```
data-type="hex:6:1:-"
```

The above example describes the same hexadecimal value as before while also specifying that it must have a separator character ('-') after each digit group, with a group size of one byte (two digits). Using separators is a common practise to make long numbers more human readable, like MAC48 hardware addresses and IPv6 addresses. The separator extension of the hex data type allows to store hexadecimal numbers in EDXML, including separators, without having to use the string data type. The following EDXML event object is valid for the data type defined above:

```
<mac>0d-1e-15-ba-dd-06</mac>
```

A colon can be used as separator, as the following example illustrates:

```
data-type="hex:6:1::"
```

The above data type allows objects like the one shown in the following example:

```
<mac>0d:1e:15:ba:dd:06</mac>
```

Note that the length of the data type, in bytes, must be a multiple of the group size. The group size, if specified, must greater than zero. Hexadecimal values must be lowercase and their length must match the length that the data type specifies, padding with zero digits as needed.

5.9.3 The geo family

The geo data type family is intended for representing WGS84[1] coordinates. Currently this family has just one member, which is `geo:point`. This data type requires its object values to consist of a latitude value followed by a comma and a longitude value. Both the latitude and the longitude must be decimal numbers having a precision of six digits after the decimal point[2]. An example of a valid `geo:point` object is shown below:

[1] The World Geodetic System, 1984 revision.

[2] This yields a precision of 20 cm at the equator, well within the limits of WGS84

```
<geolocation>43.133122,115.734600</geolocation>
```

Latitude values must be between -90.000000 and 90.000000, both inclusive. Longitude values must be between -180.000000 exclusive and 180.000000 inclusive. When the latitude value is either -90.000000 or 90.000000 the longitude value must be 0.000000. Both latitude and longitude must not have any leading zeros or '+' sign. Also, the decimal parts must have exactly six decimal digits, padded with zeros if necessary[3].

[3] These requirements guarantee that any location has only one corresponding `geo:point` value.

5.9.4 The uri data types

These data types represent URIs as defined in RFC 3986. The second component of the data types must be the path separator of the applicable URI scheme. For example, an URL data type could be defined as

```
data-type="uri:/"
```

while a URN(Uniform Resource Name) could be defined as

```
data-type="urn::"
```

The value space of the data types covers any string. Since each URI schema defines its own rules for valid URIs it is impractical to attempt to define a value space and validate them. The distinction between the `uri` data types and the `string` data types is mostly semantic.

5.9.5 The file data type

The `file` data type can be used to refer to externally stored files. The value space of the data type matches that of the data types in the `uri` family. The difference between the `uri` data types and the `file` data type is the intended use.

The `file` data type is intended for partial URIs which can be combined with externally provided configuration data to generate full URIs. This allows the referenced files to be relocated without changing the EDXML data. Two parties exchanging EDXML data may each store the referenced files on their own local infrastructure and use local configuration to expand the file references into URIs that resolve correctly in their local environment.

5.9.6 The uuid data type

This data type represents universally unique identifiers (UUID) as defined in RFC 4122 of the Internet Engineering Task Force (IETF). The values are lowercase canonical textual representations of UUIDs: 32 hexadecimal digits, in five groups separated by hyphens, in the form `8-4-4-4-12` for a total of 36 characters.

5.9.7 The ip family

The `ip` data type family consists of two members. The `ip:v4` data type is used for storing Internet Protocol version 4 (IPv4) addresses while the `ip:v6` data type is used for storing version 6 (IPv6) addresses. Values of the `ip:v4` data type must be written in dotted decimal notation, where each octet has any leading zeros removed. Example:

```
192.168.0.1
```

Values of the `ip:v6` data type must be lowercase colon separated hexadectets. All eight hexadectets must be specified, address shortening rules must not be applied. Example:

```
2001:0db8:0000:0000:0000:ff00:0042:8329
```

5.9.8 The datetime data type

This data type is used for representing date / time values. The value space is the range of valid single points in time as defined in ISO 8601, with the following EDXML specific restrictions:

- Valid years are from 1583 to 9999 (both inclusive)[1]
- The time zone must be UTC and explicitly specified as 'Z'

[1] While ISO 8601 allows years before 1583, implementations of the standard may or may not support them.

- The time value 24:00:00 is not valid
- Seconds must be specified with a precision of exactly six decimals

These restrictions guarantee compatibility with a wide range of storage systems and guarantee that every possible value can only be represented in one way: If the string representation is the same, it must be the same value. Also, the restrictions imply that date / time values can be correctly sorted using simple lexicographical string sorting.

5.9.9 The sequence data type

The `sequence` data type can store unsigned 64-bit integers. As such its value space matches that of the `number:bigint` data type. The difference between the two is in the semantics. The `sequence` data type is intended to be used by EDXML data sources that generate an incremental sequence of numbers, with or without gaps. Database record IDs that increment each time a new record is added are a good example. Line numbers in parsed logging files is another common use case.

Note that this data type can be used to determine the ordering of events, as detailed in section 2.7.1.

5.9.10 The boolean data type

The `boolean` data type can be used for boolean values. Its value space is limited to the string values “true” and “false”.

5.9.11 The enum data types

The `enum` data types can be used to create a custom value space containing a fixed set of strings. These strings can be specified in the data type by appending them as components. For example, the following enum definition allows three values ‘yes’, ‘no’ and ‘maybe’:

```
data-type="enum:yes:no:maybe"
```

5.9.12 The base64 data types

These data types represent Base64 encoded strings. They can be used to store values that may contain characters which cannot be represented as valid XML, like some control characters for example. The values of this data type must be valid `base64Binary` values as defined in the W3C XML Schema language recommendation, with the additional restriction that whitespace is not allowed. An example of a base64 data type definition is shown below:

```
data-type="base64:1024"
```

The mandatory integer component following the family name is the maximum length of the values, in bytes. A length of zero means that values can have any length except zero.

The base64 data types are intended for strings that may be displayed as text, even though some characters may not be printable. Therefore the data types should only be used for strings that contain mostly printable characters. Binary data like pictures or arbitrary files should be stored as event attachments or in externally stored files that are referenced from events.

5.9.13 The string family

The `string` data type family can be used for anything that does not suit any of the other data types mentioned above. An example of a string data type is shown below:

```
data-type="string:32:mc:ru"
```

String data types are composed of the following components:

1. The data type family: 'string'
2. The maximum length of the object values, 32 in this example. Use '0' (zero) to specify unlimited length
3. Letter case indicator
4. Modifier

The possible values of the letter case indicator and their meanings are listed below:

- `mc`: Values may contain any mix of uppercase and lowercase characters.
- `lc`: Values must contain only lower case characters.
- `uc`: Values must contain only upper case characters.

The letter case indicator can be used to enforce canonicalization of values to either pure lower case or pure upper case values. It can be used for object types that represent values where character case has no meaning, such as ISO 3166-1 country codes for example.

The maximum length can be exploited by some database systems to optimize storage and processing. It is recommended to indicate a maximum length whenever possible.

The modifier component is optional. It may contain any combination of the characters 'r' and 'u', in any order. The meaning of these characters is specified below.

u By default the value space of the string data types is limited to strings which are encoded using the 8-bit latin1 character encoding. Presence of the 'u' character in the modifier component widens the value space to strings which are encoded using the utf-8 character encoding.

r Presence of the 'r' character in the modifier component indicates that it may be advantageous for database systems to store the values in reverse character order.

As the use of the utf-8 encoding may incur additional overhead for storage and processing it is recommended to use it only when needed.

Depending on the nature of the strings and expected search patterns that may be used to search them it may or may not be beneficial to indicate reverse character order. As an example consider an object type representing phone numbers. Phone numbers may or may not include a country or area code, which makes it more practical to search them using the rightmost part. Also, the country and area codes in the leftmost part may yield low cardinality database indexes. Storing these objects in reverse order can improve index efficiency.

5.10 The fuzzy-matching attribute

The optional `fuzzy-matching` attribute can be used to indicate how to find objects that are to be considered similar. The attribute must not be specified for object types that have any data type other than those from the `string` family. The available fuzzy matching modes are specified below.

5.10.1 The phonetic matching mode

Phonetic matching indicates that objects should be considered similar when they sound similar. An object type representing names is a typical example where phonetic matching can be useful. This matching mode is selected by setting the `fuzzy-matching` attribute to "`phonetic`".

5.10.2 The head matching mode

This type of fuzzy matching compares the leftmost characters. If they are identical, the objects should be considered similar. This matching mode is selected by setting the `fuzzy-matching` attribute to `[n:]`, where `n` is the number of characters that must be identical. Example:

```
fuzzy-matching="[6:]"/>
```

5.10.3 The tail maching mode

This type of fuzzy matching compares the rightmost characters. If they are identical, the objects should be considered similar. This matching mode is selected by setting the `fuzzy-matching` attribute to `[:n]`, where `n` is the number of characters that must be identical. Example:

```
fuzzy-matching="[:6]"/>
```

Note that matching strings on their tail is an expensive operation in many database systems. These systems can often overcome this problem by storing the object values in reverse order. For this reason it is recommended to combine this fuzzy matching mode with the reverse storage indicator in the `data-type` attribute[1].

[1] Refer to section 5.9.13 for details on reverse string storage.

5.10.4 The substring matching mode

This type of fuzzy matching compares a specific part of the strings. If that part is identical, the objects should be considered similar. This matching mode is selected by setting the `fuzzy-matching` attribute to `substring:` followed by a regular expression. The first match found in the string is the part that is used in the comparison. The regular expression may optionally contain a parenthesized subexpression. In that case the part that matches the parenthesized subexpression is used in the comparison rather than the full match.

To illustrate the use of the substring matching mode we provide an example below. The example defines an email address which specifies that fuzzy matches occur when the local part of the email address is identical:

```
<object-type data-type="string:254:mc:u"
             name="computing.email.address"
             display-name-singular="email address"
             display-name-plural="email addresses"
             description="RFC 5322 email address"
             fuzzy-matching="substring:^[^@]+"/>
```

Alternatively, one might want to define a fuzzy match when the domain part

of the email address matches. This is accomplished using the following object type definition:

```
<object-type data-type="string:254:mc:u"
             name="computing.email.address"
             display-name-singular="email address"
             display-name-plural="email addresses"
             description="RFC 5322 email address"
             fuzzy-matching="substring:@(.+)"/>
```

5.11 Versioning and Upgrading

The attributes of the `object-type` tag and their upgrading details are specified in table 9. Please refer to section 6 for details about ontology upgrading.

Table 9: Upgrading of object type attributes.

attribute	upgradable
`name`	n
`description`	y
`data-type`	y*
`display-name-singular`	y
`display-name-plural`	y
`unit-name`	y
`unit-symbol`	y
`prefix-radix`	y
`xref`	y
`compress`	y
`fuzzy-matching`	y
`regex-hard`	y*
`regex-soft`	y

Some attributes are upgradable only when certain conditions are met. In table 9 these are marked with a superscript asterisk (y*). The conditions are detailed below.

regex-hard Newer object type versions must not set this attribute when it was unset in older versions. Newer object type versions may change the attribute value of the previous version by appending an alternation operator (|) followed by another regular expression.[1] This allows object type upgrades to extend the value space of the object type.

[1] Due to the alternation operator having the lowest operator precedence the regular expression engine will try to match the expression after the alternation operator in case the part before it does not match.

data-type Newer object type versions may change the `data-type` attribute in order to

extend the value space of the object type. Only those changes that are described in section 5.12 are allowed.

5.12 Data Type Upgrading

The `data-type` attribute of the `<object-type>` tag can be upgraded if and only if that upgrade is specified in the following sections.

5.12.1 Upgrading the enum data type

Newer versions of object types that use the `enum` data type family may change their data type relative to the previous version by appending one or more components to the `data-type` attribute. For example, the data type `enum:yes:no` can be upgraded to `enum:yes:no:maybe`.

6. Ontology Versioning and Upgrading

EDXML features a distributed ontology. Ontology information may be distributed across multiple `<ontology>` elements which we refer to as *ontology fragments*. These fragments may in turn be scattered across multiple EDXML documents. This allows for combining EDXML data from multiple data sources that each generate their own domain ontology. Merging multiple ontology fragments is a common operation in EDXML applications.

Each type of ontology component (object type, concept, source and event type) has an XML attribute that contains its unique identifier. Two ontology components of the same type that share a common identifier are defined to be instances of the same ontology component. The attribute containing the unique identifier of the ontology component is given for each component below:

object type	`name`
concept	`name`
event type	`name`
source	`uri`

When merging two instances of a particular ontology component there are three possible scenarios:

1. Both instances are equivalent[1]
2. One is a valid upgrade of the other
3. Neither is a valid upgrade of the other

[1] Equivalence is defined in section 12

Two ontology fragments are defined to be mutually compatible when they do not have any components in common for which the third scenario applies. Only mutually compatible ontology fragments can be merged.

This section specifies how ontology upgrades work and when two ontology components are valid upgrades of each other. While describing the various ontology components we already specified which aspects of these components can be upgraded. Upgrades are intended to allow an EDXML data source to refine a previously defined ontology element. To this end all ontology components have a numerical `version` attribute which defines their versions. Consider two instances of the same ontology component, $\mathcal{A}$ and $\mathcal{B}$. Now $\mathcal{B}$ is defined to be an upgrade of $\mathcal{A}$ when the version of $\mathcal{B}$ is greater than the version of $\mathcal{A}$. This upgrade can be either valid or invalid.

An upgrade of an ontology component is valid only then the new version introduces no changes other than those that are allowed in upgrades of that specific type of component.[2] The allowed changes for the various ontology components are detailed in their respective sections:

[2] Note that an upgrade that introduces no changes other than changing the version is also a valid upgrade.

Event types	Section 2.8 on page 32
Sources	Section 3.4 on page 35
Concepts	Section 4.5 on page 37
Object types	Section 5.11 on page 51

The allowed changes for ontology components are designed such that upgrades are always backward compatible: Any event that is valid for version v of its event type must also be valid for any version v' of that same event type when $v' \geq v$. For example, given any event $\mathcal{E}$ that is valid for version 1 of its event type $\mathcal{T}_e$ then $\mathcal{E}$ must also be valid for versions 2, 3, … of $\mathcal{T}_e$.

Backward compatibility also holds for the semantics of events. An ontology upgrade should not change the meaning of the events as defined by previous versions of the event type. As such, ontology upgrades are intended to be used to correct errors in ontologies and extend them. For example, an

upgrade might fix a spelling error in an event type description or add an optional property.

Note that an EDXML ontology as a whole has no version, only its components do. This is a result of the ontology being distributed in nature. Each EDXML data source only needs to output the event type definitions that are relevant to the data that it produces. And each data source can independently output upgrades of those event types without any coordination taking place with other data sources.

7. Foreign Elements and Attributes

Third parties may define proprietary extensions of the EDXML specification by introducing custom XML elements and attributes. We will refer to these as foreign elements and attributes. The foreign elements and attributes are intended to be used by client / server combinations that both agree to support a particular proprietary extension of the EDXML specification. As these extensions are valid EDXML, implementations of the EDXML specification must accept the extensions as valid EDXML. However, EDXML implementations may ignore foreign elements and attributes and drop them on ingestion. The RelaxNG schema allows for extensions at specific extension points which are specified below.

7.1 Foreign event attributes

The `<event>` tag accepts foreign attributes provided that the attributes have a namespace that differs from the EDXML namespace. This implies that the attributes must have a namespace prefix.

7.2 Foreign elements

The root `<edxml>` element accepts arbitrary child elements provided that the elements have a namespace that differs from the EDXML namespace. Contrary to `<event>` elements foreign elements may precede the first `<ontology>` element in an EDXML document.

8. Event Hashes

Each logical EDXML event has a unique persistent hash. This hash is not explicitly stored in the event itself. It can be computed from the event data using the procedure described in section 8.1. The hash includes a subset of the properties of an event. The properties in this subset are referred to as *hashed properties*. Hashed properties are defined as properties that have the `match` merge strategy. Depending on how the hashed properties are chosen multiple physical events may share a common hash. A set of physical events that have identical hash values represent a single logical event. As such the hashed properties determine what uniquely identifies a logical event. It also implies that the hash of a logical event is persistent. EDXML event types can be designed to allow events to be updated by producing a new physical event that has the same hash as an existing event. The hash 'sticks' to the event as it evolves over time. For this reason, EDXML event hashes are also referred to as *sticky hashes*.

8.1 Hash computation method

Sticky hashes are computed from the following three hash components:

1. The URI of the event source
2. The name of the event type
3. A subset of the event objects

Each hash component is a string. The string representation of the third component, the event objects, is specified later in this section.

To compute the sticky hash, the above three hash components must be concatenated using a linefeed character (`\n`) as separator between the components, in the given order. Then, the resulting string must be hashed using a hashing function. Sticky hashes which appear in EDXML documents for a specific intent that is described elsewhere in this specification must be computed and represented as specified for that specific intent. For all other uses of sticky hashes any hashing function and hash representation may be used as long as the hashes are sufficiently unique for the purpose.

The subset of event objects that is the input for the third hash component is selected by taking the objects of all hashed properties of the event. The string representation of the third hash component is obtained from the event object subset by means of the following steps:

1. Each event object is prepended with the name of its property and a colon (`:`).
2. Any duplicates among the prepended objects are removed, resulting in a set of unique strings.
3. The strings in the set are sorted using a binary sort.
4. The sorted set of strings is concatenated using the four byte sequence `0xffffffff`[1] as separator.

[1] This byte sequence cannot occur in any valid unicode string, which means that it is unambiguous as a separator in this context.

8.2 Example

Consider the following event representing a response from a DNS query service:

```
<event event-type="internet.dns.record.a"
       source-uri="/internet/dns/">
  <properties>
    <ip>173.194.67.104</ip>
    <ip>173.194.67.103</ip>
    <ip>173.194.67.106</ip>
    <ip>173.194.67.147</ip>
    <ip>173.194.67.105</ip>
    <ip>173.194.67.99</ip>
    <domain>www.google.com</domain>
    <queried>2012-03-11T00:00:21Z</queried>
  </properties>
</event>
```

Now assume that properties `domain` and `ip` are the hashed properties of the associated event type. This implies that there are seven objects involved in the sticky hash calculation. After prepending the property names, we get the following object substrings:

```
ip:173.194.67.104
ip:173.194.67.103
ip:173.194.67.106
ip:173.194.67.147
ip:173.194.67.105
ip:173.194.67.99
domain:www.google.com
```

Next, we sort the substrings and remove any duplicates:

```
domain:www.google.com
ip:173.194.67.103
ip:173.194.67.104
ip:173.194.67.105
ip:173.194.67.106
ip:173.194.67.147
ip:173.194.67.99
```

Concatenating the event object substrings and combining all three hash components we obtain the following input string for the hashing function:

```
/internet/dns/\n
internet.dns.record.a\n
ip:173.194.67.103\xff\xff\xff\xff
ip:173.194.67.104\xff\xff\xff\xff
ip:173.194.67.105\xff\xff\xff\xff
ip:173.194.67.106\xff\xff\xff\xff
ip:173.194.67.147\xff\xff\xff\xff
ip:173.194.67.99\xff\xff\xff\xff
domain:www.google.com
```

In the above string, only the literal `\n` are linefeed characters and the `\xff` represent the bytes of the four byte object value separators.

9. Resolving Event Collisions

A given set of EDXML documents may contain multiple physical events that represent instances of one and the same logical event. Two physical events represent a single logical event when they share the same sticky hash[1]. When this occurs the two events are said to *collide*. This enables EDXML data sources to generate updates for a previously output event. An EDXML data consumer may want to resolve collisions and merge the events. The following sections specify how that operation must be performed.

[1] See section 8 for details about sticky hashes.

9.1 Merging event objects

For each of the properties of the event type, the event objects of the colliding events are combined and merged. The `merge` attribute[2] of the event property indicates the *merge strategy* that must be used to merge the objects of that property in the colliding events. The merge strategy takes the combined objects of all colliding events as input and determines the output object(s) that will be in the merged event. For each of the possible values of the `merge` attribute the merge strategy that must be applied is specified below.

[2] The `merge` attribute is specified in section 2.2.8.

min Output the object with the smallest value.

max Output the object with the largest value.

add Output all objects after discarding any duplicates. The intent is to allow adding more objects to a previously generated event.

replace Output the objects from the last[3] colliding event. If the last event has no object for the property, no objects are output.

match Output all objects after discarding any duplicates[4]. The intent is to match events on this property.

[3] Here 'last' is the last event when the colliding events are ordered by their version (see section 2.7.2)

[4] Note that, since the property is a hashed property by definition, the objects of all colliding events are identical.

set Output all objects after discarding any duplicates. The strategy indicates the intent to have an immutable event property that may initially be empty and allow populating it once, later. Therefore the object sets of the property of the colliding events should be either empty or contain identical set of objects. Unless the event type is versioned[1] EDXML implementations are not required to check if non-empty object sets are actually identical.

any Output all objects after discarding any duplicates. The strategy indicates the intent to have an immutable event property. The object sets of the property should be identical for all colliding events. Unless the event type is versioned EDXML implementations are not required to check if this is actually the case.

[1] Conflict detection requires checking all event properties regardless of their merge strategy. See section 9.4

9.2 Merging explicit parents

Any explicit parent hashes from the `parents` attributes of the colliding events must be combined into the `parents` attribute of the merged event, discarding any duplicates.

9.3 Merging event attachments

The attachments of colliding events should be identical and event merging code may safely assume that they are. As such, merge operations can pick the attachments from any of the colliding events.

9.4 Event merge conflicts

An event merge conflict is defined as the situation that occurs when merging colliding physical events that have the same event version while having differing sets of objects for at least one of its properties.

This situation may occur when multiple systems share state while each of them can independently output state updates. Representing this state using a versioned event type introduces the possibility of conflicts. These conflicts can prevent conflicting information from going unnoticed.

Colliding events of versioned event types must be checked for merge conflicts while merging. Merge conflicts generally cannot be resolved automatically.

10. Concepts in EDXML

This section is non-normative.

This specification defines several XML elements and attributes related to EDXML concepts. This section clarifies what concepts are and how these elements and attributes interact with concepts. For full background about the idea of concepts and concept mining please refer to the website[1] of the EDXML foundation.

[1] www.edxml.org

Event types provide context for event objects, defining how these objects are related. EDXML concepts can be understood as alternative contexts for event objects. The fundamental difference is that event types have actual instances in EDXML documents (the events) while concepts do not. Concepts can be instantiated as products of a process called *concept mining*, which is a form of data analysis taking EDXML events as input. Another difference is that events have properties while concepts have attributes.

Multiple event types may contribute attributes to a particular concept by means of the `<property-concept>`[2] elements in their property definitions. These elements can be seen as a mapping for transferring objects from an event context into a concept instance context.

[2] Property / concept associations are specified in section 2.2.10.

The principal problem that a concept mining algorithm needs to solve is determining which event objects to transfer into which concept instances. When multiple properties of a given event type refer to the same concept, these do not necessarily belong to the same concept *instance*. For example, an event type may contain two properties that contain an IPv4 address while both properties are associated with a 'computer' concept. These properties may refer to one and the same computer or to two different computers. In order to enable machines to make this distinction EDXML provides the inter-concept[3] and intra-concept[4] relations.

[3] The inter-concept relation is specified in section 2.3.8.

[4] The intra-concept relation is specified in section 2.3.7.

Both types of concept relations relate the objects of two properties *within a single event* to concept instances. However, the attributes of a concept instance may also be found scattered across *multiple events* that jointly contain information about a single concept instance. Correlating events that refer to the same concept instance can be done by identifying objects shared between events. Consider two events containing the same IPv4 address and assume that the properties of both event types associate those objects with the same concept. Then, these events may either refer to the same computer or they may refer to two different computers that happen to have the same IPv4 address. In order to enable machines to make this distinction they need to know if an IPv4 address is a unique identifier of a computer or not. This is where the `confidence` attribute of the `<property-concept>` element comes in. This confidence is not a binary yes / no value, it is a confidence indicating how strong of an identifier the property is for instances of the concept.

Instantiating a concept can be implemented as an iterative process of associating one EDXML object with another. Taking one object within a specific event as a starting point ('seed') and following concept relations leads to more objects. These objects in turn lead to more events that have this object in common, and so on. Each step has an intrinsic risk of error. The errors originate from the various `confidence` attributes that exist in EDXML, as outlined below.

- Properties have confidences implying that objects may be inaccurate.
- Relations have confidences implying that following them may lead to the wrong objects.
- Property / concept associations have confidences implying that a property may be a weak concept identifier, resulting in incorrectly correlated events.

These confidences enable event type designers to integrate uncertainty as it exists in various aspects of data and enable machines to reflect that uncertainty in analysis results.

11. EDXML Templates

EDXML templates are used in various contexts to transform event information into text phrases. The most prominent use case for templates can be found in the `story` attribute of an event type definition. This template transforms the data of an event into a little story. While EDXML ontology information is primarily intended for consumption by machines, EDXML templates are specifically intended for human consumption. From an event type designer perspective it enables the designer to convey all of the intricacies of interpretation of the various event properties and how they relate to one another. From an end user perspective it enables the end user to learn the exact meaning of an event in an intuitive way.

11.1 Template syntax

EDXML templates utilize the concept of data binding: They contain placeholders which refer the properties and attachments of the event type for which they are defined. When an EDXML template is evaluated into a string the placeholders must be replaced with strings that depend on the objects and attachment content from a specific event.

Consider the following template:

```
phone call from [[caller]] to [[callee]]
```

The strings wrapped in double square brackets (blue) are placeholders. The strings between the brackets are property names. In this example, the placeholders must be replaced with the objects of the `caller` and `callee` properties. When evaluated, the template might yield the following output string:

phone call from 0034656286219 to 0034642772906

Since event properties may be optional the placeholders can refer to properties which may not have any objects. This may vary from one event to another. Since all events of a specific event type share the same template, this one template needs to evaluate into a proper text phrase for any valid event of that event type. This is achieved by creating *template scopes*.

When any of the placeholders in the template evaluates into an empty string, the full template must evaluate into an empty string. When this happens, we say that the template has *collapsed*. In practise many templates will refer to optional event properties, which means that the template will yield an empty string for some of the events. To counter this effect the collapse of the template can be contained within a part of the template by using template scopes. A template scope is a part of the template that is enclosed in a pair of curly brackets, as shown below[1]:

[1] We use the ↪ symbol to indicate line breaks in long templates. These are not part of the actual template string.

```
A phone call to [[callee]] was registered{, originating from [[
    ↪ caller]]}.
```

The curly brackets shown above (in green) define a template scope that will limit a collapse of the template to the scope enclosed in the brackets. If, in the above example, the caller property has no objects then only the scope that contains the caller property will collapse and the template still evaluates into a proper text phrase. Template scopes enable designing templates which yield strings that gracefully degrade when evaluated for events that lack objects for some of its properties.

In a slightly more advanced example, two scopes are used for events that contain objects for either of two event properties, but never both:

```
The web site can be reached by means of {IP address [[ip]]}{
    ↪ host name [[host-name]]}.
```

Template scopes may also be nested. The below example illustrates how this might be useful:

```
Client record of client number [[client-recordno]].{ The record
    ↪  contains the following additional information:{{ Date
    ↪ of birth is [[date-of-birth]].}{ Client has an order
    ↪ history of [[order-count]] orders.}}}
```

This example will produce a useful output string even when only a client number is available and all other information is missing.

EDXML templates may be further enhanced by employing *object formatters*. Object formatters are used to change how event objects translate into strings. Consider the following example:

```
On [[date_time:timestamp,minute]], a phone call took place from
    ↪  [[caller]] to [[callee]].
```

In the example the `timestamp` property contains ISO 8601 timestamps, which do not look natural to humans. The `date_time` string is a formatter which transforms ISO 8601 timestamps into more human friendly dates and times. A formatter must be followed by a colon. The string after the colon must be a comma separated list of formatter parameters. In the case of a `date_time` formatter the first parameter is a property name, the second parameter indicates the display accuracy of the timestamp. Without the formatter, the above template might evaluate to:

```
On 2010-09-17T15:14:50.000000Z, a phone call took place from
                +316385529 to +31699265109.
```

Using the formatter, the output might look more like this:

```
          On Friday, September 17th 2010 at 15:14h,
   a phone call took place from +316385529 to +31699265109.
```

Formatters can also be used to control scoped template collapse. Consider a template containing the following phrase:

```
... The shipment is located in [[city]] [[country]].
```

Now assume that both properties mentioned in the template are optional, which means they might not have any objects. We could use curly brackets to limit the scope of the template collapse:

```
...{ The shipment is located in [[city]] [[country]]}.
```

Unfortunately this has the undesired effect of collapsing the entire phrase when one property has no objects while the other does. Now consider the following alternative approach:

```
... The shipment is located in {[[city]]} {[[country]]}.
```

Now, when both properties are empty the phrase evaluates into a malformed string:

```
                The shipment is located in .
```

In situations like this the `unless_empty` formatter can be used. It evaluates into a string constant unless all of the specified event properties are empty. Consider:

```
...{ The shipment is located [[unless_empty:city,country,in]] {
    ↪ [[city]]} {[[country]]}.}
```

With this formatter in place and both properties empty, the formatter will yield an empty string causing the scope that contains it to collapse. When any of the two properties has an object the formatter will yield "in" and template collapse is prevented.

11.2 Template formatters

A small variety of formatters is available for use in EDXML templates. Each will be described in the following sections. Note that these sections purposely do not always specify every single detail of how the formatters should be implemented. For that reason the evaluation results shown in the examples are not normative. Evaluated templates may vary slightly between different implementations of the specification and between various target output mediums.

Contents

11.2.1 The date_time formatter

The `date_time` formatter generates a string representing the date and time of an object having the `datetime` data type. It has two parameters. The first parameter is the property name. The second parameter indicates the display accuracy of the value. It accepts one of the following values:

year	Show year
month	Show year and month
date	Show year, month and day (date)
hour	Show year, month, day and hour
minute	Show year, month, day, hour and minute
second	Show year, month, day, hour, minute and second
millisecond	Show year, month, day, hour, minute, second with millisecond precision
microsecond	Show year, month, day, hour, minute, second with microsecond precision

Example:

```
It happened in [[date_time:date,month]].
```

which might evaluate into:

It happened in March 2015.

11.2.2 The duration formatter

The `duration` formatter computes a duration from two objects having the `datetime` data type. It takes two property names as parameters. Example:

```
It took [[duration:begin,end]].
```

which might evaluate into:

It took 6 hours, 3 minutes and 34 seconds.

11.2.3 The time_span formatter

The `time_span` formatter translates two timestamps into a description of the time period in between these two timestamps. It takes two property names are parameters. Example:

```
It happened [[time_span:begin,end]].
```

which might evaluate into:

> *It happened on January 12th 2017 between 13:43h and 16:31h.*

11.2.4 The url formatter

The `url` formatter renders objects as a hyperlinks, if the target output medium supports this[1]. It features two parameters. The first is the name of an event property that contains the URLs. The second parameter is a string containing a name for the URL, which is intended for generating clickable hyperlinks that display the name of the URL rather than the URL itself. Example:

[1] An EDXML template evaluator outputting to HTML for instance could support this.

```
More information can be found [[url:external-ref,on this
    ↪ website]].
```

An evaluator targeting an HTML output medium might evaluate this into:

```
<p>More information can be found <a href="http://some.site">on
    ↪ this site</a>.</p>
```

A renderer targeting a plain text output medium in stead might evaluate this into:

> *More information can be found on this site (http://some.web.site).*

11.2.5 The merge formatter

The `merge` formatter merges the objects from one or more properties into a single list of objects. The formatter takes any number of property names as arguments. Example:

```
The order consists of [[merge:books,magazines]].
```

which might evaluate into:

> *The order consists of The Da Vinci Code and National Geographic.*

11.2.6 The boolean_string_choice formatter

The `boolean_string_choice` formatter yields one out of a pair of two strings, depending on the value of an object having the `boolean` data type. It requires three parameters. The first parameter is the name of an event property. The second parameter is a string which is rendered for object value `true`. The third parameter is a string which is rendered for object value `false`. Example:

```
The server has a [[boolean_string_choice:host.ip.is-public,
    ↪ public,private]] IP address.
```

which might evaluate into:

The server has a public IP address.

11.2.7 The boolean_on_off formatter

The `boolean_on_off` formatter yields either "on" or "off", depending on the value of an object having the `boolean` data type. It requires the name of a property containing boolean objects as its only parameter. Example:

```
The alarm was switched [[boolean_on_off:alarm-status]].
```

which might evaluate into:

The alarm was switched off.

11.2.8 The boolean_is_is_not formatter

The `boolean_is_is_not` formatter yields either "is" or "is not", depending on the value of an object having the `boolean` data type. It requires the name of a property containing boolean objects as its only parameter. Example:

```
The alarm [[boolean_is_is_not:alarm-status]] activated.
```

which might evaluate into:

The alarm is activated.

11.2.9 The empty formatter

The `empty` formatter yields a fixed string in case a property has no objects, while producing an empty string in all other cases. It requires the name of an event property as its only parameter. Example:

```
The file{, which has a file type of [[file.type]],}{, [[empty:
    ↪ file.type,of which no file type is known]],} was
    ↪ received on [[date_time:timestamp,seconds]].
```

Depending on the presence / absence of objects the above template might evaluate into:

The file, which has a file type of application/pdf,
was received on March 3rd 2008 at 23:54:12h.

or

The file, of which no file type is known,
was received on March 3rd 2008 at 23:54:12h.

11.2.10 The unless_empty formatter

The `unless_empty` formatter yields a fixed string unless none of the specified properties have any objects. It requires the names of one or more event properties as parameters, followed by a final parameter containing the fixed string. Example:

```
The shipment is ready{ and is located [[unless_empty:city,
    ↪ country,in]] {[[city]]} {[[country]]}}.
```

When a country object is available but no city object this template might evaluate into:

The shipment is ready and is located in the Netherlands.

When no country object and no city object is available this template will evaluate into:

The shipment is ready.

11.2.11 The attachment formatter

The `attachment` formatter renders an event attachment into the output string rather than event objects. Its only parameter is the name of an attachment that is defined for the event type. As attachments are typically

used to store relatively long string or binary values, EDXML implementations may or may not render them in line. Depending on the attachment media type, output medium and other factors they may also be rendered as a separate paragraph, as an image, or in some other form. For this reason it is recommended to position any place holders containing this formatter at the end of a sentence. An example is shown below.

```
The message text is: "[[attachment:message]]"
```

Above template might evaluate into:

The message text is:
"Hello World"

12. Equivalence

This specification uses the notion of *equivalence* in various places. In this section we define equivalence for EDXML events, ontology components and EDXML documents.

Consider two EDXML documents $\mathcal{A}$ and $\mathcal{B}$. These documents are defined as each others equivalents when the following two conditions are met:

- Each logical event in $\mathcal{A}$ must have an equivalent event in $\mathcal{B}$, and the other way around.
- The combination of all `<ontology>` elements from $\mathcal{A}$ must be the equivalent of the combination of all `<ontology>` elements from $\mathcal{B}$, and the other way around.

Note that the logical events can be obtained from an EDXML document by merging any colliding physical events as specified in section 9.

12.1 Event Equivalence

To define the meaning of equivalence of two EDXML events it is useful to define the *normal form* of an EDXML event first:

Normal form of an EDXML event

An `<event>` element in Canonical XML[1] form representing a valid EDXML event where any missing optional attributes are set to their defaults and which has its objects and attachments sorted as follows:

- The objects must be sorted on property name, then on the value string.
- The attachments must be sorted on attachment name, then on their `id` attribute value.

The hashes in the `parents` attribute of the `<event>` tag must be sorted lexicographically.

[1] Canonical XML Version 2, as specified by the W3C

Two EDXML events are defined to be each others equivalents when their normal forms are identical.

12.2 Ontology Equivalence

Given two EDXML ontologies $\mathcal{A}$ and $\mathcal{B}$ we define $\mathcal{A}$ and $\mathcal{B}$ to be each others equivalents when each of the object types, concepts, event types and sources in $\mathcal{A}$ has an equivalent definition in $\mathcal{B}$ and the other way around. We define equivalence for object types, concepts, event types and sources below.

12.2.1 Object Type Equivalence

To define the meaning of equivalence of two object type definitions it is useful to define the *normal form* of a object type definition first:

Normal form of an object type definition

An `<object-type>` element in Canonical XML[2] form representing a valid object type where any missing optional attributes are set to their defaults.

[2] As specified by the W3C

Two object type definitions are defined to be each others equivalents when their normal forms are identical.

12.2.2 Concept Equivalence

To define the meaning of equivalence of two concept definitions it is useful to define the *normal form* of a concept definition first:

Normal form of a concept definition

A `<concept>` element in Canonical XML[3] form representing a valid concept where any missing optional attributes are set to their defaults.

[3] As specified by the W3C

Two concept definitions are defined to be each others equivalents when their normal forms are identical.

12.2.3 Source Equivalence

To define the meaning of equivalence of two source definitions it is useful to define the *normal form* of a source definition first:

Normal form of a source definition

A `<source>` element in Canonical XML[1] form representing a valid source where any missing optional attributes are set to their defaults.

[1] As specified by the W3C

Two source definitions are defined to be each others equivalents when their normal forms are identical.

12.2.4 Event Type Equivalence

To define the meaning of equivalence of two event type definitions it is useful to define the *normal form* of an event type definition first:

Normal form of an event type definition

An `<event-type>` element in Canonical XML[2] form representing a valid event type where:

[2] As specified by the W3C

- any missing optional attributes are set to their defaults,
- its properties, relations and attachments are sorted,
  - the properties and attachments by their names,
  - the relations by their type, then by their `source` attribute and finally by their `target` attribute,
- the property mappings listed in the `property-map` attribute of the `<parent>` element, if present, are sorted.

Two event type definitions are defined to be each others equivalents when their normal forms are identical.

13. Tag and Attribute Reference

This section offers a quick reference to all XML tags and attributes defined by the EDXML specification. For each tag, it lists their parent tag and attributes.

Tag	Parent	Attribute	Description
edxml			Root tag
		version	EDXML version
ontology	edxml		Contains ontology fragment
event-types	ontology		Contains event type definitions
event-type	event-types		Event type definition
		name	Name of event type
		display-name-singular	Display name (singular form)
		display-name-plural	Display name (plural form)
		description	Description
		summary	Short event description template
		story	Full event description template
		timespan-start	Time span start property
		timespan-end	Time span end property
		sequence	Logical event sequence number
		event-version	Event version
		version	Event type version
parent	event-type		Parent definition
		event-type	Event type of parent
		property-map	Child / parent mapping
		parent-description	Child / parent relation description
		siblings-description	Child / sibling relation description
properties	event-type		Event type properties element
property	properties		Property definition
		name	Name
		description	Description
		object-type	Associated object type
		similar	Event similarity hint
		confidence	Object confidence
		merge	Merge strategy
		optional	Optional property flag
		multivalued	Multivalued property flag
relations	event-type		Property relations element
intra	relations		Intra-concept relation definition
		source	Property name (source)
		target	Property name (target)
		source-concept	Concept name (source)
		target-concept	Concept name (target)
		description	Description of relation
		predicate	Relation predicate
		confidence	Relation confidence
inter	relations		Inter-concept relation definition
		source	Property name (source)

Tag	Parent	Attribute	Description
		target	Property name. (target)
		source-concept	Concept name (source)
		target-concept	Concept name. (target)
		description	Description of relation
		predicate	Relation predicate
		confidence	Relation confidence
name	relations		Name relation definition
		source	Property name (source)
		target	Property name. (target)
description	relations		Description relation definition
		source	Property name (source)
		target	Property name. (target)
container	relations		Container relation definition
		source	Property name (source)
		target	Property name. (target)
original	relations		Original relation definition
		source	Property name (source)
		target	Property name. (target)
other	relations		Generic relation definition
		source	Property name (source)
		target	Property name (target)
		description	Description of relation
		predicate	Relation predicate
		confidence	Relation confidence
property-concept	property		Property / concept association
		name	Concept name (target)
		confidence	Concept identifier confidence
		cnp	Concept naming priority
		attr-extension	Attribute name extension
		attr-display-name-singular	Attribute display name (singular form)
		attr-display-name-plural	Attribute display name (plural form)
attachments	event-type		Event attachment definitions
attachment	attachments		Event attachment definition
		name	Name of the attachment
		description	Description of the attachment
		display-name-singular	Display name of the attachment (singular form)
		display-name-plural	Display name of the attachment (plural form)
		media-type	RFC 6838 media type of the attachment
		encoding	Encoding of the attachment
object-types	ontology		Contains object types
object-type	object-types		Object type definition
		name	Name of object type

Tag	Parent	Attribute	Description
		display-name-singular	Display name of object type (singular form)
		display-name-plural	Display name of object type (plural form)
		description	Description of object type
		data-type	An EDXML data type
		unit-name	Unit of measurement (name)
		unit-symbol	Unit of measurement (symbol)
		prefix-radix	Radix of metric prefixes
		xref	Reference to additional details
		compress	Compression hint
		regex-hard	Regular expression (hard)
		regex-soft	Regular expression (soft)
		fuzzy-matching	Fuzzy comparison option
		version	Version
concepts	ontology		Contains concept definitions
concept	concepts		Contains concept definition
		name	Concept name
		display-name-singular	Display name of concept (singular form)
		display-name-plural	Display name of concept (plural form)
		description	Description of concept
		version	Version
sources	ontology		Contains sources
source	sources		Source definition
		uri	Source URI
		description	Description of source
		date-acquired	Acquisition date in yyyymmdd format
		version	Version
event	edxml		Event
		event-type	Name of event type
		source-uri	Source URI
		parents	List of parent hashes
properties	event		Event properties
attachments	event		Event attachments
		id	Event attachment identifier