EDXML

Specification Document

edxml.org

version 2.1.0

Contents

1. Introduction

Today's corporations and organizations often have to face the challenges induced by data fragmentation: Information is no longer contained in a one information system, but data lives in many different places. These isolated information islands make it increasingly hard to really know what is going on. Data integration projects try to interconnect all of these data islands, attempting to reveal the big picture and enable global data analysis.

This document describes the EDXML information carrier. EDXML is an XML based data carrier, used to communicate a broad scope of different kinds of information between a variety of systems. Due to this broad scope, it can facilitate data integration projects.

EDXML is more than just another canonical data reprentation though. It integrates data with semantics, which can significantly reduce the complexity and cost of data integration projects, while providing greater flexibility to adjust your systems to future changes in your organization. The semantics that is integrated in EDXML data describe the logical structure of the data and provide hints to aid analysis tasks. In stead of having to program your systems to handle 25 diffent types of data from 12 different sources, information systems can automatically learn how to perform analysis, visualization and reporting of the data. Most - if not all - of the required application logic can be dynamically derived from the EDXML data streams themselves. With this feature, it is even possible to develop generic data processing systems that can process *any* EDXML data, without any prior knowledge about the exact nature of the data. As a result, these data processors tend to be relatively simple and do not need to be upgraded before any new types of information can be introduced into your systems, reducing both development and maintenance cost.

To further enhance flexibility, designing EDXML data representations does not require that any assumptions be made about the data storage and processing systems that will be used. Weather you plan to use Python scripts, SQL database systems or massive Hadoop clusters to process your data, it makes no difference to the EDXML representation.

The semantics are encoded in the *ontology* that is present in any EDXML stream. The ontology describes the structure of the data in the stream, in terms of properties and objects. EDXML streams do not need to contain an ontology that is complete, in the sense that it describes all data that could possibly be contained in EDXML streams. It is possible to include only the part of the ontology that is required to interpret the data in that particular EDXML stream. In the presence of multiple EDXML generating information systems, each system only needs to generate a partial *domain ontology*. EDXML allows various compatible domain ontologies to be combined into a more complex ontology, by merging streams from various sources. This kind of distributed ontology can have various advantages over a single large and complex ontology. For example, upgrading the domain ontology in one information system can often be done without touching any of the other information systems.

Despite the fact that EDXML can be used to represent a wide range of types of data, it is not a very complex format. It is composed of no more than 18 different tags. In fact, its simplicity is one of its key features. The EDXML effort includes keeping the specification as clean and simple as possible, to ensure that generating, interpreting and processing EDXML data streams remains simple enough to allow agile development of efficient data processing solutions.

The following sections describe the various aspects of the EDXML specification in detail. The Software Developers Kit (SDK) contains a reference implementation of the specification, which can be used as a jump-start base for experimenting with generating and processing EDXML data. The reference implementation is written in Python, but the specification itself is language independent and is easily implemented in any general purpose programming language.

2. Valid EDXML

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

1. The EDXML data validates against the RelaxNG schema included in the EDXML SDK
2. The EDXML data adheres to the specifications outlined in this document.

The reason for splitting up the definition of valid EDXML data is twofold. First of all, XML schemas are too limited to encompass the expressiveness of EDXML, mainly due to limited support for cross-reference checking. This means that an XML schema alone will not suffice to define what a valid EDXML document is. And second, omitting any kind of XML schema would require describing the rules imposed by the RelaxNG schema in this specification document, which would not improve its readability.

Please refer to section 4.1 for details about validating EDXML data.

3. Changes

Version 2.1 of the EDXML specification contains a number of enhancements over version 2.0, as well as preparations for EDXML 3.0. Note that all changes are downward compatible with EDXML 2.0. All changes are listed below.

- Property relations can now be defined to be unidirectional or bidirectional.
- Object values can automatically be validated against a regular expression, by means of the new `regexp` attribute of the `objecttype` tag.
- A new `parent` tag has been added, which allows an event type to have an implicit parent event type. Implicit parent definitions can be used to infer explicit parent-child relationships between events.
- New data types have been added for storage of numerical values in hexadecimal representations as well as for storing geolocations.
- The `property` tag has a new optional `similar` attribute, which can be used to hint EDXML implementations on how to find similar events.
- The `fuzzy-matching` attribute of the `objecttype` tag has a default value: '`none`'. This value is the equivalent of not specifying the `fuzzy-matching` attribute at all. Also, a new fuzzy matching method is introduced, which is a generalization of the existing head/tail matching methods. It allows using regular expressions to match specific patterns within strings.
- Both the `eventtype` and the `objecttype` tags now have an optional `display-name` attribute. This attribute is used to define a short, human friendly name for event types and object types.
- `objecttype` tag now features an optional `compress` attribute, which can be used to hint EDXML storage backends on the expected gains of object value compression. Also, a `display-name` attribute was added for providing human friendly object type names.

- Object types that are preferred for naming entities can now be hinted using the new `enp` attribute of the `objecttype` tag.
- Events can now specify explicit parent events, by means of the new optional `parents` attribute of the `event` tag.
- Some new reporter string formatters have been added.
- Three new event merging strategies have been added which allow arithmatic operations to be performed on event properties while merging.

Great care has been taken to implement these changes in the EDXML SDK while maintaining backward compatibility as much as possible. Still, existing software built on top of version 2.0 of the SDK may require minor updates when upgrading the SDK.

Note that this version of the specification features the description of the new method for computing sticky hashes, which will replace the current method in version 3 of the specification. This allows for preparations to ease the transition to EDXML 3.0. Even though this specification document decribes it, the new hashing method is explicitly *not* part of EDXML 2.1.

4. General EDXML structure

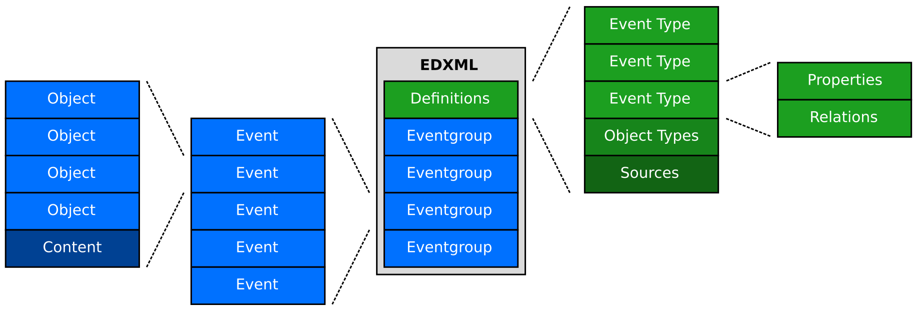
EDXML represents all data by means of *events*. The name 'event' suggests that it represents an occurrence / incident and that an event should always have an associated time stamp. An event can be used to represent much more than just occurrences though. Depending on your application, events can also be regarded as documents (e-discovery), or traces (forensic science). An EDXML event really is quite an abstract concept, which is probably best described as follows:

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

In other words, an event is just a container which groups some loose bits of information together.

In this specification document, we will frequently use phone call records as examples of events. A phone call event contains a set of information elements, like the phone numbers involved, a time stamp and a duration. These information elements are called objects. 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.

The general layout of an EDXML stream is shown below. Every EDXML stream or file consists of two parts. One part contains the actual data, represented by events (blue). The format of this part of the EDXML data is very simple, it's basically just a list of objects grouped into events. The other part of the EDXML data, shown in green, is a lot more complex. This header, which precedes the event data, contains all of the information that is required to interpret the events. The content of this header is often referred to as the dataset ontology, which describes the structure of all event types, object types, and so on. The ontology also includes *semantics*. Semantics in EDXML is basically just hints for machines, which explain how to interpret and analyze the information contained in the events. For example, an event type definition may contain hints explaining how objects within an event relate to each other, and how the event data can be used to perform - for example - social network analysis (SNA).



Due to the simplicity of the representation of the actual event data on the one hand and the more complex definitions section on the other, EDXML has sort of a dual character. Parsing out the event data, feeding it into a large NoSQL document store, or converting it into a spreadsheet document is very easy to do: You don't need the complex definitions part (green) of the EDXML files to do this sort of thing, you can just ignore it. Parsing out the simple, flat event data (blue) is all you need to do. On the other hand, the context and semantics contained in the definitions part can be used to generate complex data structures which may, for instance, be used for graph analysis or data mining.

4.1 Validating EDXML

Many of the features of EDXML cannot be easily expressed in terms of a hierarchical XML structure. Therefore, implementing the specification into an XSD or RelaxNG schema is hard or even impossible. This does not mean that EDXML files cannot be validated though. The SDK contains a RelaxNG schema which can validate the general structure of EDXML. The SDK contains a reference implementation of an EDXML validator which validates all aspects of the specification that the RelaxNG schema cannot validate. It offers both a commandline EDXML validator and classes which can be used to develop EDXML processors with integrated input and output validation.

Even though a generic, complete XSD or RelaxNG schema for EDXML does not exist, it is possible to *generate* both XSD and RelaxNG schemas based on a given EDXML ontology which is known to be valid. These generated schemas allow for relative validation, which checks if the content of one EDXML file is compatible with the content of another EDXML file. This is extremely valuable, because compatible EDXML files can be merged. So, a set of compatible EDXML files is effectively a single, consistent, distributed dataset. This means that relative validation can for instance be used to develop EDXML generators which validate their output against the content of a central EDXML data store.

The SDK contains modules which can generate schemas for relative validation. Also, the EDXML validator in the SDK features relative validation capability.

The next EDXML release is expected to allow a greater part of the validation to be done using XML schemas.

4.2 Events and Event Groups

EDXML files use the UTF-8 character encoding, and the entire document is enclosed in a `events` tag. The events element must contain two sub-elements: a `definitions` element, followed by an `eventgroups` element. This layout is shown below.

```
<events>
<definitions>
...
</definitions>
<eventgroups>
...
</eventgroups>
</events>
```

For brevity, the content of the `definitions` and `eventgroups` elements have been omitted in the above XML structure. The dots (...) which replace the omitted parts in the above example will be used consistently throughout this document to indicate omitted parts in XML syntax examples.

The definitions element contains definitions of each of the event types used in the EDXML data, which we will describe in detail in section 5. The `eventgroups` element contains zero or more `eventgroup` elements, which contain events that share a particular combination of event type and source. This layout is shown below.

```
<eventgroups>
<eventgroup event-type="phonecall" source-id="1"/>
<eventgroup event-type="phonecall" source-id="2"/>
...
</eventgroups>
```

The eventgroup elements contain the actual data, the events. Each eventgroup must have an event type and a source ID set. The event type determines the event type of all events in the eventgroup. The source ID refers to a source definition in the definitions section of the EDXML stream, as we will see later on. Every eventgroup can contain zero or more events. The events themselves look like this:

```
<event>
<object property="caller" value="0034656286219"/>
<object property="callee" value="0034642772906"/>
<object property="duration" value="5"/>
</event>
```

Here, `property` must be assigned the name of a property of the event type of the eventgroup, as defined in the definitions section, while `value` is the actual object value. Generally, for every property of the event type, one may specify zero or more objects. There are exceptions when certain merge strategies are used, refer to section section 5.2.5 for details.

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

Events can have optional content. Content is a body of plain text, which is typically used in event types describing email messages, content extracted from a PDF document, Twitter messages and so on. We can extend the above phone call event to have content, like this:

```
<event>
<object property="caller" value="0034656286219"/>
<object property="callee" value="0034642772906"/>
<object property="duration" value="5"/>
<content>Hello? Who is calling?<content/>
</event>
```

Events can also have one or more translated versions of their content:

```
<event>
<object property="caller" value="0034656286219"/>
<object property="callee" value="0034642772906"/>
<object property="duration" value="5"/>
<content>Hello? Who is calling?<content/>
<translation language="nl" interpreter="john">Hallo? Met
    wie?<translation/>
</event>
```

The language must be an ISO 639-1 language code. The interpreter value can be used to indicate the source of the translation. It may be the name of an interpreter, a number, anything you want.

The translation tag is deprecated in EDXML 2.1 and will be removed in EDXML 3.0.

The definitions section defines all event types that are used in the event groups, source definitions, and object type definitions. The general structure looks like this:

```
<definitions>
  <eventtypes>
    <eventtype>
    ...
    </eventtype>
  </eventtypes>
  <objecttypes>
    <objecttype>
    ...
    </objecttype/>
  </objecttypes>
  <sources>
    <source>
    ...
    </source>
  </sources>
</definitions>
```

For brevity, the content of the `eventtype`, `source` and `objecttype` elements are omitted in the above XML structure. Their content is described in more detail in the next section.

5. Event types

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

5.1 Defining an event type

An abbreviated example of a phone call event type definition is given below:

```
<eventtype name="communication-phonecall"
           display-name="phone call/phone calls"
           description="help desk phone call record"
           classlist="helpdesk"
           reporter-short=... reporter-long=... >
  <property name="communication-phonecall-caller"
            description="caller"
            object-type="communication-phonenumber"/>
  <property name="communication-phonecall-callee"
            description="callee"
            object-type="communication-phonenumber"/>
  <property name="communication-phonecall-duration"
            description="duration"
            object-type="time-duration"/>
  <relations>
    <relation property1="communication-phonecall-caller"
              property2="communication-phonecall-callee"
              description=...
              type="inter:contacted"
              confidence="0.9"
              directed="true"/>
  </relations>
</eventtype>
```

The event type itself has six attributes: `name`, `display-name`, `description`, `reporter-short` and `reporter-long`. You can probably guess what the name and description attributes are for. The `display-name` attribute was introduced in EDXML v2.1 and is optional. It exists of two parts, separated by a slash. The part left of the slash contains the singular form, the right part contains the plural form. Compared to the name attribute, it allows a wider range of characters to be used (like spaces and capitals), and features both singular and plural forms. For this reason, the display-name attribute is useful in generating human readable content in print, graphical user interfaces, and so on. Omitting this attribute is equivalent to setting it to its default value: '/'.

The `classlist` attribute provides a comma separated list of event type classes that the event type belongs to. One event type can belong to multiple event type classes and one event type class may contain multiple event types. Event type classes offer a means to refer to multiple event types using a single identifier.

The reporter strings (`reporter-long` and `reporter-short`) require a bit more explanation. Reporter strings are templates that can be used to construct a textual representation of the information in the event, in human readable form. It places the object values in the context of an event type. Reporter strings utilize the concept of data binding: They contain special placeholders, which refer to objects in the event. These placeholders can be replaced with the object values of a specific event, which yields a human readable description of that event. Short reporter strings are meant to be used whenever space is limited, and may mention just one or two objects from the event. Long reporter strings are supposed to provide a complete textual representation of all of the objects in an event.

In the phone call example, one might use a short reporter string that looks like this:

```
phone call from [[communication-phonecall-caller]] to
[[communication-phonecall-callee]]
```

The property names between the double square brackets are the placeholders for the object values. When evaluated, the resulting short event description might look like this:

phone call from 0034656286219 to 0034642772906

When any of the properties referred to in the reporter string does not have an object, the reporter string must evaluate into an empty string. As a consequence of this, depending on how your event types are defined, constructing a reporter string containing all event properties may often result in an empty string. For this reason, reporter strings have a feature which allows certain parts of the string to be omitted, depending on the presence or absence of objects. Let us look at a simple example first:

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

The curly brackets in the above reporter string indicate that this substring is optional. It is omitted when no object of the communication-phonecall-caller property exists in the event. The curly brackets can also be used in cases where you know that either one of two event properties can have an object, but never both. Example:

```
The server can be reached by means of{ IP address [[
   server-ip]]}{ host name [[server-hostname]]}.
```

Substrings may also be nested. Another example to illustrate how this might be useful:

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

The above example will produce a proper description, even when it contains only a client number. The produced reporter string will gracefully degrade as the amount of available information decreases.

When evaluating reporter strings containing substrings, the following rule must be applied in an iterative fashion to each substring, and *only* to substrings, starting at the innermost substring:

when substrings are present, and all of them yield an empty string, an empty string results.

The above rule should always be kept in mind when designing reporter strings. As a final example, we will look at a reporter string which describes a location event in varying degrees of detail. It will show a country, optionally followed by a region and in some cases the event also contains the name of a city. One might be tempted to use the following reporter string:

```
The location is in the country with country code [[
    internet-geoip-country]]{, in the region of [[
    internet-geoip-region]]{ and in the city named [[
    internet-geoip-city]]}}.
```

This reporter string will not work correctly though. Let us apply the above processing rule to this reporter string. Processing always begins at the innermost substring, containing the city name in this example. Suppose that there is no city name in the event. The substring containing the city property will evaluate to an empty string. The city substring is contained in the region substring. If there is no city object, all substrings the region substring will evaluate into an empty string. According to the processing rule, this means that the region substring itself will also yield an empty string, *even when a region object exists*. In the end, the absence of a city object results in the following evaluated event description:

The location is in the country with country code us.

Note that, had we applied the substring processing rule to the reporter string as a whole, the result would have been empty. The processing rule is only to be applied to substrings, though. We can improve our example reporter string by making sure that there is at least one non-empty substring left in case the city substring processes empty. The final processing string which yields the desired result looks like this:

```
The location is in the country with country code [[
    internet-geoip-country]]{, in the region of {[[
    internet-geoip-region]]}{ and in the city named [[
    internet-geoip-city]]}}.
```

Now, the region substring will not yield an empty string if there is no city object in the event.

Placeholder strings may be further enhanced by means of formatters. Formatters are used to transform object values into more human friendly strings. Consider the following example:

```
On [[FULLDATETIME:communication-phonecall-timestamp]], a
    phone call took place from [[communication-phonecall
   -caller]] to [[communication-phonecall-callee]].
```

Here, the `communication-phonecall-timestamp` property contains UNIX timestamps, which most humans are not very comfortable with. The `FULLDATETIME` string is a *formatter*, separated from the property name by a colon. The `FULLDATETIME` used in this example transforms UNIX timestamps into human readable dates and times. Without the formatter, the above reporter string would generate strings like this:

On 1284736490, a phone call took place from +316385529 to +31699265109.

Using the formatter, the output might look like this:

On Friday, September 17th 2010, at 15:14:50h UTC,
a phone call took place from +316385529 to +31699265109.

The EDXML specification defines a small variety of formatters, as listed below.

DATE	Generates a string representing the date part of a UNIX timestamp
DATETIME	Generates a string representing the date and time of a UNIX timestamp
FULLDATETIME	A more elaborate version of DATETIME
WEEK	Translates a UNIX timestamp into a week number and year
MONTH	Translates a UNIX timestamp into a month and year
YEAR	Translates a UNIX timestamp into a year
DURATION	Computes a duration from two timestamp properties. Two comma seperated properties are specified, like [[DURATION:timestamp-begin,timestamp-end]]
TIMESPAN	Translates two timestamps into a description of the amount of time between these two timestamps. Two comma seperated properties are specified, like [[TIMESPAN:timestamp-begin,timestamp-end]]

LATITUDE	Translates a floating point or decimal property into degrees, arc minutes and seconds
LONGITUDE	Translates a floating point or decimal property into degrees, arc minutes and seconds
BYTECOUNT	Translates a integer property into byte units, like 10 KiB, 32 MiB
CURRENCY	Formats a floating point or decimal property as a sum of money, like $427.56. The currency symbol is specified like [[CURRENCY:transaction-sum:$]]
COUNTRYCODE	Transforms a ISO 3166-1 alpha-2 country code into a full country name.
FILESERVER	Transforms the object value into a URL. As an example, you could use this formatter on hashlinks, producing HTML event descriptions that contain clickable links to files stored on a file server.
BOOLEAN_STRINGCHOICE	Displays one of two strings, depending on the value of a boolean property. The strings are supplied like [[BOOLEAN_STRINGCHOICE:client-is-male:Mr.:Ms.]]
BOOLEAN_ON_OFF	Like BOOLEAN_STRINGCHOICE, but uses fixed strings 'on' (true) and 'off' (false)
BOOLEAN_IS_ISNOT	Like BOOLEAN_STRINGCHOICE, but uses fixed strings 'is' (true) and 'is not' (false)
EMPTY	Produces a fixed string in case the property has no objects, while producing an empty string in all other cases. An example is given below.

Note that, within the square brackets of the placeholders, colon characters (:) may only be used to separate the formatter from the property name(s). Also, comma characters (,) may not be used, unless the formatter requires it to be used as specified above.

The `EMPTY` formatter can be used to produce an output string, even if the property has no objects. It may be used in a reporter string like this:

```
The file{, which has a file type of [[file-mimetype
   ]],}{, [[EMPTY:file-mimetype:of which no file type is
    known]],} was received on [[DATE:file-timestamp]].
```

In the above reporter string, either one of the substrings in curly brackets will yield an output string, but never both.

The EDXML specification does not specify *exactly* how the above formatters should transform object values. Some formatters, like the FILESERVER formatter, are even supposed to allow your event descriptions to be customized to fit in a specific networking environment. The above list merely expresses what results EDXML designers can expect from the various formatters. Therefore, the output of generated event descriptions may vary slightly between different implementations of the specification.

5.2 Adding properties

Let's return to our initial example of an event type definition:

```
<eventtype name="communication-phonecall"
           display-name="phone call/phone calls"
           description="help desk phone call record"
           classlist="helpdesk"
           reporter-short=... reporter-long=... >
  <property name="communication-phonecall-caller"
            description="caller"
            object-type="communication-phonenumber"/>
  <property name="communication-phonecall-callee"
            description="callee"
            object-type="communication-phonenumber"/>
  <property name="communication-phonecall-duration"
            description="duration"
            object-type="time-duration"/>
  <relations>
    <relation property1="communication-phonecall-caller"
              property2="communication-phonecall-callee"
              description=...
              type="inter:contacted"
              confidence="0.9"
              directed="true"/>
  </relations>
</eventtype>
```

The property definitions all have a name, description and object type attribute. The property name should be kept short. The descriptions should describe which part of the event is represented by the property. Short descriptions like 'caller' suffice. Because any property has a context (which is the event type), machines can generate more verbose property descriptions when needed. For example, combining a property description with the event type description, computers can describe the same 'caller' property as 'caller in a phone call event'.

If two properties have the same name attribute, all of the other attributes of both properties must be identical as well. Two different event types may define properties having the same name attribute.

The object type of a property is the name of the object type of the objects, which must be defined in the `<objecttypes>` section. We will see how object types are defined later on. Besides name, description and object type, properties can have a few more attributes that you can specify in a property definition. Each of these will be addressed below.

5.2.1 defines-entity

The `defines-entity` attribute is used to indicate if the property contains identification information of an entity. Examples of entity defining properties could be phone numbers (like the caller in a phone call), or email addresses (like the sender of an email message). For these properties, the `defines-entity` attribute should be set to "true". If the `defines-entity` attribute is set to "false", or if it is omitted, the property must not be regarded as entity identifying information. Even though the attribute is optional, it is recommended to always specify it when generating EDXML streams, as it will make future migration to EDXML v3 easier.

Entity defining properties can be used by analysis systems as starting points or building blocks for social network analysis, or for profiling of entities. Please refer to section 11 for more information about the concept of entity in EDXML.

5.2.2 entity-confidence

The `entity-confidence` attribute is used to express the *entity identifying authority* of a property. Entity identifying authority expresses how authoritative a property is for identifying an entity. More precise:

The probability that the correct entity is selected from all entities in the dataset, when the only selection criterion is the object value of the property for which the authority is being defined.

For example, a social security number has a high entity confidence, because this number should be fairly unique. On the other hand, a family name should have a low confidence, because there could be many different entities in your dataset having the same family name. The `entity-confidence` is a floating point number between 0.0 and 1.0. Determining the best suitable value should not be taken to be an exact science. A rough estimate (like 0.8) will do. Analysis systems can use this confidence value to estimate error margins of analysis results.

This attribute is optional, omitting it is equivalent to specifying a confidence of zero. However, it is recommended to always specify this attribute when generating EDXML streams, as it will make future migration to EDXML v3 easier.

5.2.3 similar

This 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. The `similar` attribute allows analysis systems to explicitly prepare specific queries and describe to the user what the query will do. To this end, the `similar` attribute should contain a short phrase that describes how one event relates to similar events that share the same property value. The description should fit into a grammar construct like this:

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

For instance, when a `similar` attribute value of *"originating from"* is used for property `communication-phonecall-caller` of event type `communication-phonecall` will result in a query description like this:

Find all phone calls originating from the same caller

This attribute is optional, omitting it is equivalent to specifying an empty attribute. However, it is recommended to always specify this attribute when generating EDXML streams, as it will make future migration to EDXML v3 easier.

5.2.4 unique

An event type can be unique, which means that there can be only one event of that event type which has a specific combination of properties. An event type is unique if it has at least one unique property, which is a property that has its `unique` attribute set to "true". Typical examples of unique properties are unique identifiers from database systems, like a client record ID or a help desk support ticket ID. This attribute is optional, when omitted it is to be regarded as "false". However, it is recommended to always specify this attribute when generating EDXML streams, as it will make future migration to EDXML v3 easier.

It should be noted that unique properties can never have more than one object in any single event.

5.2.5 merge

Events can update each other by means of *event collisions*. Suppose that an information system contains event A. Next, it receives event B on its input. Now assume that both events have the same unique event type. Events A and B are said to collide if the object values of all of their unique properties are identical. Information systems can resolve event collisions by using the *merge strategy* indicated by the `merge` attribute. The merge strategy indicates how two colliding events should be merged into one. Unique properties *must* have the `merge` attribute set to `match`, which implies that the property value must match for two events to collide. For all other properties, the attribute is optional. If an optional merge attribute is not specified, the merge strategy for that property defaults to `drop`, which implies that the property should be ignored in merge operations. If specified, the `merge` attribute must have one of the values listed below. For each value, the merge strategy is described, where the 'source' is a new event that collides with an existing event called the 'target'.

add	Any new object from the source event will be added to the target event.
replace	If the target event already has an object for this property, it will be replaced by the object of the source event, unless the source has no object, in which case the object in the target is deleted. If the target lacks an object, the object in the source event will be added, if any.
drop	Any objects of this property in the event that is to be merged, will be ignored.
min	The existing object will be replaced with the object that has the lowest numerical value.
max	The existing object will be replaced with the object that has the highest numerical value.
increment	The target object value will be incremented, regardless of the value of the source object.
sum	The target object will be replaced with the sum of both the source and target object values.
multiply	The target object will be replaced with the product of the source and target object values.
match	No action. Unique properties, and only unique properties, must use have this merge strategy.

Merge strategies allow for a EDXML generating data source to update existing events in other EDXML enabled information systems, which makes EDXML suitable for live, dynamic applications like system monitoring. Event merging can also be used for aggregating data from many separate events into one, or be used to implement simple distributed NoSQL-style map-reduce schemas.

Properties that use the `match`, `min`, `max`, `increment`, add or `multiply` strategy, must have exactly one object value. Properties that use the `replace` strategy must have either zero or one object value.

Properties that use the `min`, `max`, `increment`, add or `multiply` strategy, must have an object type that has a data type that

- is a member of the `number` family, but not `number:hex`, or
- is the `timestamp` data type.

Please refer to section 7.1 for details about data type families.

Note that the `increment` strategy effectively counts the number of times an event has been merged with another event.

Although the `merge` attribute is optional, it is recommended to always specify this attribute when generating EDXML streams, as it will make future migration to EDXML v3 easier.

5.3 Adding property relations

After defining the properties of an event type, one may optionally specify relations between these properties. Let's return to our initial example of an event type definition once again:

```
<eventtype name="communication-phonecall"
           display-name="phone call/phone calls"
           description="help desk phone call record"
           classlist="helpdesk"
           reporter-short=... reporter-long=... >
  <property name="communication-phonecall-caller"
            description="caller"
            object-type="communication-phonenumber"/>
  <property name="communication-phonecall-callee"
            description="callee"
            object-type="communication-phonenumber"/>
  <property name="communication-phonecall-duration"
            description="duration"
            object-type="time-duration"/>
  <relations>
    <relation property1="communication-phonecall-caller"
              property2="communication-phonecall-callee"
              description=...
              type="inter:contacted"
              confidence=0.9
              directed="true"/>
  </relations>
</eventtype>
```

This example contains one property relation definition, between the "communication-phonecall-caller" property and the "communication-phonecall-callee" property. The definition also specifies a description, type, confidence and if the relation is directed or undirected.

The description attribute contains a template, similar to the reporter string used in the `eventtype` tag. The template must contain two placeholders, one for each of the two related properties. 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>

The placeholders must consist of the names of the two related properties, enclosed in double square brackets. 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 [[communication-phonecall-caller]] and [[communication-phonecall-callee]]

The type attribute consists of two parts separated by a colon. An example is shown below:

```
inter:is acquainted to
```

The part left of the colon must be one of the five relation classes that exist in the EDXML specification:

inter	This type of relation connects one entity to another. In the phone call example, the relation between caller and callee should be defined as being of the 'inter' type.
intra	This type of relation connects information about the same entity. It may, for instance, relate an email address to the name of its owner.
parent	Used to represent parent-child relationships between properties *(deprecated)*.
child	Identical to parent, but the direction of the relation is inverted *(deprecated)*.
other	Used for any other kind of relation

Relations of the `inter` type can be used for social network analysis. Jumping between objects through a property relation of the `inter` type leads to other, related entities.

Relations of the `intra` type can be used for entity profiling. Jumping between objects through a property relation of the `intra` type leads to related information about the same entity.

The right part of the type string ("is acquainted to") is a free form predicate which can be used to characterize the nature of the relation. The purpose of the predicate is to facilitate generating triples, like the ones found in RDF triple stores, of the form

<subject> <predicate> <object>

The predicates should be as short as possible, and it is advised to use just a small set of different predicates in your dataset. For example, the following predicates

- is married to
- communicates with
- is called

could be processed into the following triples:

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

Apart from materializing relations for feeding triple stores or graph databases, predicates can also be useful for the following tasks:

Filtering	Generate a graph of events that are related by means of relations having a specific predicate.
Labeling	Predicates can be used to indicate the nature of links between nodes in a relation graph, by printing the predicates next to the links.
Descriptions	Triples can be used whenever a short description of a relation is needed, for example in places where space is limited.

The `confidence` attribute of a property relation is defined as follows:

The probability that the relation actually exists when it is observed once.

Let us clarify this by means of the phone call example. In the example, we defined a relation between the caller and callee, specifying relation predicate ”is acquainted to”. If we have *one* event where Alice phones Bob, we know that Alice and Bob will probably be acquainted to each other. However, Alice might have dialed the wrong number, and might not know Bob at all. This is the relation uncertainty that is expressed by the confidence value. The ‘confidence’ is a floating point number between 0.0 and 1.0. Determining the best suitable value should not be taken to be an exact science. A rough estimate (like 0.8) will do. Please note that the confidence expresses the confidence associated with a *single* observation. For example, when the dataset contains 25 phone calls between Alice and Bob, the odds that Alice dialed the wrong number are getting pretty slim. Due to the way relation confidences are defined, analysis systems can take observation counts into the equation to compute net confidences of observation ensembles.

Relation confidences can also be used by machines to estimate the confidence of indirect relations. Suppose that an analysis system discovers that objects A and C are related, because object A is related to B, and B is related to C. The combination of multiple relation confidences can be used to estimate the net confidence of the relation between A and C.

The `parent` and `child` relation types are deprecated in favour of the new `<parent>` tag, which allows superior means of expressing parent-child relationships. Both types will be removed in EDXML v3.

The `directed` attribute was introduced in EDXML v2.1. It is optional, defaults to ”true”, and indicates if the relation should be regarded as directed or not. Directedness is an edge property in graph theory. When a set of EDXML relations are combined to form a graph, the directedness attribute can be used to create both directed and undirected edges between nodes. If `directed` is set to ”true”, the direction of the edge points from `property1` to `property2`.

5.4 Adding a parent definition

The deprecated relation classes `parent` and `child` have a replacement in EDXML 2.1. Since this version of the specification, parent-child relations can be specified in two ways, explicitly and an implicitly. The implicit parent-child relation specifies how to find the parents of events of a specific event type. The explicit parent-child relation is a one-on-one relation between two events, as detailed in section 8.5. Implicit parent specifications are part of the event type definition. For this purpose, an optional `<parent>` element can be added. This element must be the first element in a `<eventtype>` block as shown in the below example:

```
<eventtype name="communication-phonecall" ... >
  <parent eventtype="client-subscriptionrecord"
          propertymap="phonecall-
              clientid:subscriptionrecord-clientid"
          parent-description="associated with"
          siblings-description="related to"/>
  ...
</eventtype>
```

In the above example, we express the fact that every phone call event is related to an event describing the subscription details of the client that made the call. We do that by defining a parent for the ‘communication-phonecall’ event type, in this example an event of type ‘client-subscriptionrecord’. Next, we link the two event types by means of properties containing the client identifier. Both event types have a property containing the client identifier. So, given any phone call event, we can now find its associated subscription record by finding the event of type ‘client-subscriptionrecord’ of which the value of the ‘subscriptionrecord-clientid’ property matches the value of the ‘phonecall-clientid’ 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.

In the parent definition shown above, the `eventtype` attribute specifies the name of the event type of the parent. The event type of the parent must be defined in the same EDXML stream. The `propertymap` attribute links child properties to matching properties in the parent. The `propertymap` attribute must contain a link to each of the unique properties of the parent. Consequently, every child event has exactly one parent.

Event types that define a parent must have no more than one object for each property in the `propertymap` attribute of the parent definition. Also, each child property in the `propertymap` must have a merge strategy of either `match` or `drop`. These restrictions assure that events have one fixed parent event which cannot be changed by merging it with another event. Note that it is possible to omit an object for a property that is mapped to a parent property, but only when its merge strategy is `drop`. In that case, the parent of the event cannot be determined. When the parent has multiple unique properties, multiple property mappings must be specified by creating a comma separated list, as follows:

```
propertymap="<childproperty1>:<parentproperty1>,<
    childproperty2>:<parentproperty2>,..."
```

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

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

The parent relation description can be used by computers to generate text phrases like:

"Click here to view the client subscription associated with this phone call."

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

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

The parent relation description can be used by computers to generate text phrases like:

"Click here to view all phone call events related to the same client subscription."

6. Event Sources

A source definition looks roughly like this:

```
<source source-id="12"
        url="/company/offices/..."
        description="client records from database X"
        date-acquired="20100128"/>
```

The source ID is used to relate event groups that are defined in the same EDXML file to a source definition. For this purpose, eventgroup tags have a `source-id` attribute. Source IDs can be numbers or strings and must be unique within the scope of a single EDXML data stream.

The URL is used to provide a coarse hierarchical representation of the source. An example of a source URL is given below:

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

The URL always starts and ends with a slash. The description attribute can be used to provide more details about the origin of the data. The `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'.

7. Object types

Every property in an event type definition needs to specify its object type. Object types allow computers to learn, among other things, the meaning of 'equal'. In our phone call example, both the 'caller' and the 'callee' properties have the same object type: 'communication-phonenumber'. This is how computers will know that identical objects of the 'caller' and 'callee' properties should be considered to be the same thing.

For every object type specified in the property definitions of an EDXML file, there must be a matching definition in the object types section of that same EDXML file. An example of an object definition is shown below:

```
<objecttype name="internet-host-ipv4"
            display-name="IPv4 address/IPv4 addresses"
            description="internet IPv4 address in dotted
                decimal notation"
            data-type="ip"
            enp="40"
            compress="true"/>
```

The object type definitions all have a name, description and data type. The use of the name and description attributes is similar to the way they are used in the eventtype tag. The other attributes will be introduced below.

- The `display-name` attribute was introduced in EDXML v2.1 and is optional. It exists of two parts, separated by a slash. The part left of the slash contains the singular form, the other part contains the plural form. Compared to the name attribute, it allows a wider range of characters to be used (like spaces and capitals), and features both singular and plural forms. For this reason, the display-name attribute is useful in generating human readable content in print, graphical user interfaces, and so on. Omitting the attribute is equivalent to specifying its default value: '/'.
- The `compress` attribute was also introduced in EDXML v2.1. It is optional, defaults to "false", and 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 specific objects.
- The `enp` attribute was also introduced in EDXML v2.1. It is an optional integer in the range [0,255], defaults to zero and contains the *Entity Naming Priority* (ENP) of the specified object type. The ENP is used whenever a choice must be made between different object types to use as a name for an entity. For example, systems will prefer using a `person-name` object as the name of an entity over using a `communication-emailaddress` object if the `person-name` and

communication-emailaddress have ENP values of 200 and 100 respectively. Please refer to section 11 to learn more about the concept of *entity* in EDXML.

- The optional `regexp` attribute (not shown in the above example) has been introduced in EDXML v2.1. If specified, the attribute must contain a regular expression that is valid according to the W3C XML Schema standard. If the `regexp` attribute is specified, all object values of the object type must match the expression. Omitting the attribute is equivalent to specifying its default value, '`[\s\S]*`', which matches any string. Matching must be done on the normalized string. For example, if the data type is case insensitive, the regular expression must be matched with a lowercase version of the object value. The `regexp` attribute only applies to object types having data type `string`. For object types having other data types, this attribute must be ignored.

The role of the `data-type` attribute is explained below.

7.1 Data Types

The `data-type` attribute determines how the object values should be processed and stored by EDXML processing and storage systems. They specify details like case sensitivity, precision, character encoding, and so on. The specification provides a number of data types, which are subdivided into families and members. Each of which is described below.

7.1.1 number

The *number* data type family exists 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	32-bit floating point value
double	64-bit floating point value
decimal	exact, numerically stable fixed-point value
hex	fixed length values in hexadecimal notation

The various number data types must be specified as shown in the example below:

```
data-type="number:bigint"
```

All number data types are unsigned by default, and for every family member, except for `hex`, there is a signed version.

```
data-type="number:bigint:signed"
```

Note that the use of floating point values may cause trouble due to their numerical instability. Processing EDXML streams containing floating point values may inadvertently change the string representation of the values in the XML data, simply due to conversion to binary form. This in turn may lead to unexpected behavior, for example when events are merged. It is recommended to use the decimal data type in stead of float or double whatever possible. Refer to section 10 to learn why.

The decimal type must be specified with the desired precision, like this:

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

The two numbers shown in the data type specify the total number of digits (7) and the number of digits behind the decimal point (2). So, in the example, the number must have at most five digits before and two digits after the decimal point. Decimal numbers also have a signed variant:

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

The hex type can be used to for representing values in hexadecimal notation, with an optional separator character. It is typically used for large numbers which are generally not subject to arithmatic operations like addition or multiplication. Well known examples are hashes, IPv6 addresses and MAC addresses. In its simplest form, a hex data type attribute may look like this:

```
data-type="number:hex:12"
```

The above example describes a hexadecimal number written in 12 hexadecimal digits, which corresponds to 6 bytes. The hex data type may be extended with a separator, like this:

```
data-type="number:hex:12:2:-"
```

The above example describes the same hexadecimal number as before, but also specifies that it must have a separator character ('-') after each group of 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 you to store hexadecimal numbers in EDXML, including separators, without having to use the string data type. The following EDXML object matches the above data type definition:

```
<object property="internet-host-mac"
        value="0d-1e-15-ba-dd-06"/>
```

Using a colon as separator is also possible:

```
data-type="number:hex:12:2::"
```

This will allow you to write MAC addresses in EDXML like this:

```
<object property="internet-host-mac"
        value="0d:1e:15:ba:dd:06"/>
```

Note that the total number of digits in a hex data type must be a multiple of the digit group size.

7.1.2 geo

The `geo` datatype is a family of data types that contain geographical coordinates. At the moment, this family has just one member, which is `geo:point`. This data type exists of a latitude value, a comma and a longitude value. Both the latitude and the longitude are decimal numbers having a precision of six digits after the decimal point, which yields a precision of 20 cm at the equator. This precision is well within the limits of WGS84, the most widely used geodetic coordinate system. Example:

```
<object property="internet-ip-geolocation"
        value="43.133122,115.734"/>
```

7.1.3 hashlink

The hashlink data type contains SHA1 hashes of files stored outside of the EDXML file. Hashlinks can be used to make EDXML events refer to binary files like photographs or executables. When combined with `FILESERVER` formatters in event reporter strings, computers can generate event descriptions containing clickable links that point to files on a file server.

7.1.4 ip

This data type is used for IPv4 addresses only. IPv4 addresses can be stored as 32-bit integers, which can be exploited by EDXML data stores to efficiently store the values.

7.1.5 timestamp

Use this data type, and *only* this data type for storing times and dates. Values must be UNIX timestamps, with a maximum precision of six decimal digits. The precision is part of the specification, because the sticky hash computation method requires a consistent string representation of timestamps. Refer to section 8.2 for more information about sticky hashes. Note that UNIX timestamps are in UTC time by definition.

7.1.6 boolean

Data type which can only store the string values ”true” and ”false”.

7.1.7 enum

The enum data type can only store a fixed number of predefined strings. The enum data type takes these strings as arguments. For example, the following enum definition allows three values 'yes', 'no' and 'maybe':

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

7.1.8 string

The `string` data type 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:ci:ru"
```

The above data type definition exists of multiple components, separated by colons. The first three components are mandatory and have the following meaning:

1. The data type (string)
2. The maximum length of the object values, 32 in this example. Use '0' (zero) to specify unlimited length.
3. Indicates if the object values are case insensitive. Valid values are 'cs' to indicate case sensitivity and 'ci' to indicate case*in*sensitivity.

The maximum length can be exploited by some database systems to optimize storage and processing, it is strongly recommended to indicate a maximum length whenever possible. The third component, 'ru' in this case, is optional. It may contain any combination of the characters 'r' and 'u', in any order.

Presence of the 'u' character indicates that the object values may contain characters which cannot be represented in the 8-bit latin1 character encoding. Using 8-bit encodings for string representation in database systems is often more efficient than unicode. This hint allows those systems to take advantage of efficient one byte per character representation. When the 'u' character is present, data stores *must* use unicode to store the object values.

Presence of the 'r' character indicates that it may be advantageous to store the object values in reverse character order. There are several reasons why reverse storage may be a good idea. First, strings of which the leftmost part tends to be similar, like file paths or URLs, can lead to low cardinality database indexes. Second, matching the rightmost part of a string is inefficient in many database systems. So, strings that will generally be matched on their rightmost part may yield performance problems. As an example, you may want to indicate reverse storage for file names if you expect to do a lot of searches based on filename extension. Another example is an object type storing phone numbers. Phone numbers may or may not include a country or area code, which makes it more practical to find them by searching for the rightmost part of a phone number. Also, the country and area codes in the leftmost part may cause low cardinality indexes. Storing these objects in reverse order enables many database systems to process them more efficiently.

7.1.9 binstring

A variation of the string data type, which indicates that the string is to be treated as an array of bytes. The strings have no character set or case sensitivity. An example is shown below:

```
data-type="binstring:255:r"
```

This definition is similar to the `string` data type, but it has just two mandatory components:

1. The data type (binstring)
2. The maximum length of the object values, 255 in this example. Use '0' (zero) to specify unlimited length.

The third component, 'r' in this case, is optional. It indicates that reverse storage may be advantageous.

7.2 The fuzzy-matching attribute

The `fuzzy-matching` attribute can be used to indicate how to group object values by similarity. The available fuzzy matching techniques are described below. At the moment, all available matching techniques are applicable only to strings (the string and binstring data types).

phonetic	Phonetic matching indicates that object values should be considered similar when they sound the similar. This kind of fuzzy matching is most useful for things like names. To enable it, set the `fuzzy-matching` attribute to ”phonetic”.
match head	This type of fuzzy matching compares the leftmost characters of two objects. If they are identical, the objects should be considered to be similar. The number of characters that must match are configurable. To enable this kind of matching, set the `fuzzy-matching` attribute to `[n:]`, where `n` is the number of characters that must be identical.
match tail	This type of fuzzy matching compares the rightmost characters of two objects. If they are identical, the objects should be considered to be similar. The number of characters that must match are configurable. To enable this kind of matching, set the `fuzzy-matching` attribute to `[:n]`, where `n` is the number of characters that must be identical. 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, this fuzzy matching technique is often combined with the reverse storage indicator in the `data-type` attribute.
match substring	A `fuzzy-matching` attribute consisting of ”substring:” followed by a regular expression indicates that a fuzzy match is found when two object values both contain a match against the regular expression *and* both matches are identical. The regular expression may optionally contain a parenthesized subexpression. In that case, only the parts of the object values that match the parenthesized subexpression must be identical for a fuzzy match to occur.
none	Using a `fuzzy-matching` attribute of ”none” indicates that no fuzzy matching is applicable.

The attribute is optional and has a default value of ‘none’. It is recommended to always specify this attribute, as it will make future migration to EDXML v3 easier.

To illustrate the use of the fuzzy-matching option, we provide four examples below. The first example defines an email address, which triggers fuzzy matches when the local part of the email address matches:

```
<objecttype data-type="string:254:cs:u"
            name="communication-emailaddress"
            display-name="email address/email addresses"
            description="internet email address"
            compress="true"
            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:

```
<objecttype data-type="string:254:cs:u"
            name="communication-emailaddress"
            display-name="email address/email addresses"
            description="internet email address"
            compress="true"
            fuzzy-matching="substring:@(.+)"/>
```

The next example defines a phone number, which triggers fuzzy matches when the last six characters match and indicates reverse storage:

```
<objecttype data-type="string:32:ci:r"
            name="communication-phonenumber"
            display-name="phone number/phone numbers"
            description="telephone number"
            fuzzy-matching="[:6]"/>
```

The last example defines a name of a person, which triggers fuzzy matches when the name sounds similar:

```
<objecttype data-type="string:1024:ci:u"
            name="person-name"
            display-name="name/names"
            description="name or alias of a person"
            fuzzy-matching="phonetic"/>
```

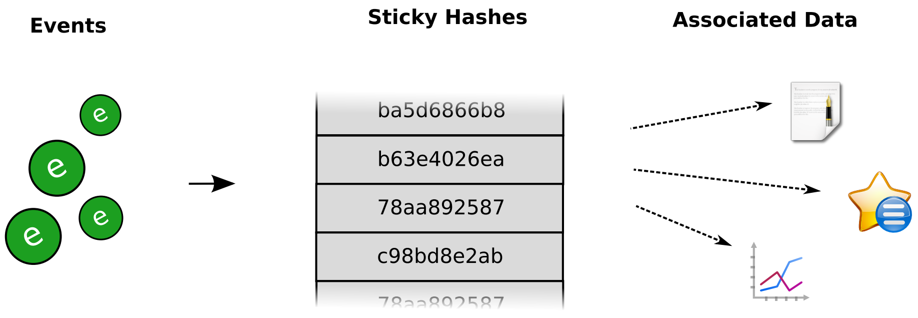
8. EDXML Sticky Hashes

8.1 What sticky hashes are all about

The EDXML specification does not allow for ad-hoc properties to be added to individual events. The reason for this limitation is the context based nature of EDXML. All events of a specific event type have the same context (the event type *is* the context). Properties without context have no meaning, therefore they cannot exist outside of the context of the event type. Consequently, all events have the same set of properties.

Still, many analysis en processing applications require that ad-hoc information can be associated with individual events. For example, analysts might want to associate annotations or tags with events. Or you might want to process EDXML events using a text mining application, which needs to associate ad-hoc key-value pairs to events.

The limitation mentioned above is lifted by incorporating a hashing method in the specification. Every event has a unique, associated hash value which can be used to implicitly link arbitrary information to these events in a consistent manner, as shown below.



The hashes are called *sticky* because they remain the same, even when changes are made to the events. Consider the situation where an information system contains a link between the sticky hash of event *x* and an annotation created by an analyst. Now, when event *x* changes due to an update, the link with the annotation remains intact, because the hash value does not change.

The hashing method exploits the uniqueness feature to generate sticky hashes. The object values of a unique event may change, but not all of them. The objects of unique properties are static. The hashing method requires that, for a unique event, only the objects of unique properties are used to compute the hash value. This guarantees that the hash 'sticks' to the event, even when the event is updated.

8.2 Hash computation method

The event hashes are computed using the SHA1 hashing function. The hashing function takes a utf-8 encoded string as its input. The string is composed out of three components:

1. The name of the event type, followed by a linefeed character (`\n`)
2. Object string (see below)
3. Content string (optional, see below)

For event types that have *no* unique properties, component 2 is constructed as follows. For every object, a substring is constructed by prefixing the object value with the name of its property and a colon (`:`). Next, all duplicate substrings are removed, sorted in lexicographical order and joined with linefeed characters (`\n`) in between. For a unique event, the construction of component 2 is slightly different: Only object values of unique properties are included in the object string.

Object values of `float` and `double` data types are *never* used in the hash computation.

Note that, for some data types, there is more than one possible string representation of the object value. This is why we will explicitly specify how to normalize and convert values of these data types into strings, below.

number:int	All integer values must have any leading zeros or '+' signs removed.
number:decimal	Decimal values must have any leading zeros or '+' signs removed. The fractional part (right of the decimal dot) is to be padded with zeros to match the number of digits specified in the data type.
number:hex	Hexadecimal values must be converted to lowercase. Any leading zeros must be present, such that the length of the string representation of the value matches the length as specified in the data type, separators excluded. Also, if a separator is present, any leading zeros in the individual digit groups must be present.

timestamp	Timestamps must have any leading zeros removed and must be padded with zeros to the full precision of 6 decimal digits.
geo:point	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.
boolean	String representations of booleans must be lowercase, "true" or "false".
ip	IP addresses must be represented in dotted decimal notation, where each octet has any leading zeros removed.
string	Case insensitive strings must be explicitly converted to lowercase.

The third component of the input string, the content string, is not included for unique events. Other events must include it, by appending a linefeed character (`\n`) followed by the event content.

When implementing systems which store EDXML data, it is recommended to apply the above normalization to each input event, calculate the sticky hash, use this hash to check for a possible collision with a previously stored event and merge the event with the colliding event, if any.

8.3 Example

Let us have a look at an event of the following event type, the slightly abbreviated definition of which is shown below:

```
<eventtype reporter-short=...
           classlist="dns,open sources"
           reporter-long=...
           description="DNS A record"
           name="internet-dnsrecord-host">
  <properties>
    <property object-type="internet-host-name"
              name="internet-dnsrecord-request-host"
              defines-entity="false" merge="match"
              unique="true" description="requested name"
                 />
    <property object-type="internet-host-ipv4"
              merge="match" unique="true"
              description="IP" defines-entity="false"
              name="internet-dnsrecord-ip"/>
    <property object-type="timestamp"
              name="internet-dnsrecord-timestamp-first"
              defines-entity="false" merge="min" unique=
                 "false"
              description="time of DNS lookup"/>
  </properties>
  <relations>
    <relation property1="internet-dnsrecord-request-host
       "
              property2="internet-dnsrecord-ip"
              description=...
              type="other:resolves to"
              confidence="1.00"/>
  </relations>
</eventtype>
```

Now consider the following event:

```
<event>
  <object property="internet-dnsrecord-request-host"
          value="www.google.com"/>
  <object property="internet-dnsrecord-ip"
          value="173.194.67.104"/>
  <object property="internet-dnsrecord-ip"
          value="173.194.67.103"/>
  <object property="internet-dnsrecord-ip"
          value="173.194.67.099"/>
  <object property="internet-dnsrecord-ip"
          value="173.194.67.106"/>
  <object property="internet-dnsrecord-ip"
          value="173.194.67.147"/>
  <object property="internet-dnsrecord-ip"
          value="173.194.67.105"/>
  <object property="internet-dnsrecord-ip"
          value="173.194.67.99"/>
  <object property="internet-dnsrecord-timestamp-first"
          value="1331424021"/>
  <object property="internet-dnsrecord-timestamp-last"
          value="1331429048"/>
</event>
```

Since both the property `internet-dnsrecord-request-host` and the property `internet-dnsrecord-ip` are unique properties, there are seven objects involved in the sticky hash calculation. Due to the presence of unique properties in the event definition, the event type itself is unique, which means that the event content - which is an empty string in this example - is ignored. The above event yields the following object substrings:

```
internet-dnsrecord-request-host:www.google.com
internet-dnsrecord-ip:173.194.67.104
internet-dnsrecord-ip:173.194.67.103
internet-dnsrecord-ip:173.194.67.099
internet-dnsrecord-ip:173.194.67.106
internet-dnsrecord-ip:173.194.67.147
internet-dnsrecord-ip:173.194.67.105
internet-dnsrecord-ip:173.194.67.99
```

Now we normalize the object values according to the normalization rules:

```
internet-dnsrecord-request-host:www.google.com
internet-dnsrecord-ip:173.194.67.104
internet-dnsrecord-ip:173.194.67.103
internet-dnsrecord-ip:173.194.67.99
internet-dnsrecord-ip:173.194.67.106
internet-dnsrecord-ip:173.194.67.147
internet-dnsrecord-ip:173.194.67.105
internet-dnsrecord-ip:173.194.67.99
```

Note that the string representation of one IP address changed due to the normalization process. Next, we sort the substrings and remove any duplicates:

```
internet-dnsrecord-ip:173.194.67.103
internet-dnsrecord-ip:173.194.67.104
internet-dnsrecord-ip:173.194.67.105
internet-dnsrecord-ip:173.194.67.106
internet-dnsrecord-ip:173.194.67.147
internet-dnsrecord-ip:173.194.67.99
internet-dnsrecord-request-host:www.google.com
```

Note that one substring was removed due to de-duplication. Thus, the input string for the SHA1 hash function is as follows:

```
internet-dnsrecord-host\n
internet-dnsrecord-ip:173.194.67.103\n
internet-dnsrecord-ip:173.194.67.104\n
internet-dnsrecord-ip:173.194.67.105\n
internet-dnsrecord-ip:173.194.67.106\n
internet-dnsrecord-ip:173.194.67.147\n
internet-dnsrecord-ip:173.194.67.99\n
internet-dnsrecord-request-host:www.google.com
```

Note that the last object value has no trailing linefeed. Computing the SHA1 hash of the above string yields the sticky hash:

```
541e980bddbd9b7da64b3a59755d609bb2595c13
```

8.4 EDXMLv3 hash computation method

It has been pointed out that the current hash computation method does not include information about the source of the event. This implies that two identical events from different sources yield the same sticky hash, which results in a hash collision. As any event in EDXML has only one source, it is unclear how the events should be merged. Regardless of how the events are merged, part of the information about their origins will be lost.

In order to correct this problem in the current hash computation method, EDXML version 3.0 will introduce a new hash computation method, which includes the source URL of the event. This means that events that originate from different sources will no longer yield the same sticky hash. The new hashing method will only be used from

EDXML versions starting at 3.0, but we will already describe the new method in this document.

The change that will be introduced with EDXML version 3.0 is that the source URL, along with a linefeed character, is prepended to the SHA1 input string. The input string now consists of four components:

1. The URL of the event source, followed by a linefeed character (`\n`)
2. The name of the event type, followed by a linefeed character (`\n`)
3. Object string (as in EDXMLv2)
4. Content string (optional, as in EDXMLv2)

Now let us compute the sticky hash of the same example event as shown on the preceding section, now using the EDXMLv3 method. Suppose that the event originates from a source with the following source URL:

```
/company/offices/germany/stuttgart/
```

Prepending the source URL and a linefeed character, the input string for the SHA1 hash function is as follows:

```
/company/offices/germany/stuttgart/\n
internet-dnsrecord-host\n
internet-dnsrecord-ip:173.194.67.103\n
internet-dnsrecord-ip:173.194.67.104\n
internet-dnsrecord-ip:173.194.67.105\n
internet-dnsrecord-ip:173.194.67.106\n
internet-dnsrecord-ip:173.194.67.147\n
internet-dnsrecord-ip:173.194.67.99\n
internet-dnsrecord-request-host:www.google.com
```

which yields the following sticky hash:

```
a5ad458de2b193b1737e602f0c4143d29910a4fc
```

The SDK contains reference implementations of both sticky hashing algorithms.

8.5 Explicit event parent specifications

Sticky hashes can be used to explicitly specify that event A is the parent of event B. This can be done by adding the optional *parents* attribute of the `<event>` tag, like this:

```
<event parents="a5ad458de2b19...">
  <object property="internet-dnsrecord-request-host"
          value="www.google.com"/>
  <object property="internet-dnsrecord-ip"
          value="173.194.67.104"/>
</event>
```

This method of specifying parent events is typically used when input events are processed, yielding output events. The parent attribute allows any output event to be tracked back to its origin. Also, given an event, one can find all output events that have been derived from that specific event.

The parents attribute may contain multiple sticky hashes, separated by commas. When events are merged, their parent hashes must be combined.

8.6 Merging event content and translations

At this time, the specification does not support alteration of event content or translations by merging events together. While merging a source event into a target event, any content and translations in the source events must be ignored.

9. EDXML Compliant Systems

The goal of the EDXML specification is to facilitate demanding data integration efforts. Practical solutions that use EDXML data representations typically involve a number of EDXML data sources, EDXML data processing systems and a data store. In order for the designers of each of these types of information systems to know what to expect from a data source, processing system or data store, it is useful to specify some basic properties that these systems should have. Systems that adhere to these specifications may be called EDXML compliant.

9.1 Data Sources

Except for the fact that EDXML compliant data sources must produce valid EDXML data, there are no other specifications that these systems must adhere to. There are some recommendations though.

- EDXML sources should attempt to produce many more events than event groups. Theoretically, one could place each output event into its own event group. Besides the fact that this yields a particularly inefficient EDXML representation, it also defeats some efficiency optimization opportunities for EDXML processing systems.
- EDXML sources should identify themselves by means of the source URLs of the source definitions in the generated EDXML streams. The source URL is the most coarse indication of the origin of the data in the EDXML specification, which is why it is generally a good idea to incorporate a source identifier in source URLs.

9.2 Data Stores

9.2.1 Transparancy

EDXML compliant data stores must be able to reproduce any valid input EDXML event, and each output event must be equivalent to the original input event. An input and output event are said to be equivalent when:

- The event content and translations in both events are identical
- The explicit parent hashes in both events are identical and equal in number
- The normalized string representations of all object values in both events are identical.

Note that the ordering or properties in an event is irrelevant and need not be preserved. The term *normalized string representation* refers to the normalization rules specified in section 8.2. From this definition of equivalency, it follows that the sticky hashes of the reproduced output events are identical to the sticky hashes of the original input events. It is recommended for EDXML data stores to normalize all input data according to section 8.2.

There is one exception to the transparancy rule, involving colliding events. This is explained in section 9.2.3.

9.2.2 Validation

EDXML compliant data stores must implement both absolute and relative validation. Absolute validation involves checking the input data against the EDXML specification, while relative validation involves checking an input ontology against the existing ontology that is already stored in the system. Each EDXML data stream contains a full ontology that describes the data it contains: the definitions of all event types, object types and sources. The ontology stored in new input data may conflict with the ontology that is already stored in the storage system. Data stores must check if the ontology contained in an input EDXML data stream is compatible with the ontologies of all previously stored EDXML input streams and reject the input when incompatibilities are detected.

In order to define when two EDXML ontologies are considered compatible, it is convenient to introduce the concept of the *Canonical Form* of EDXML. Canonical Form EDXML is EDXML data which

- has been converted to Canonical XML
- has all optional tag attributes set to their default values
- has its property tags ordered by their name attribute.

Here, 'Canonical XML' is the recommendation from the World Wide Web Consortium (W3C). Now, we will define when two EDXML ontologies are considered compatible.

Given two Canonical Form EDXML files, the ontologies of these EDXML files are compatible if the `eventtype`, `objecttype` and `source` elements that are shared between both EDXML files are identical.

When an ontology element is 'shared between both EDXML files', we mean that both EDXML files contain an `eventtype`, `objecttype` or `source` element having the same `name` attribute.

Note that EDXML v3 is expected to introduce ontology versioning, which means that ontology incompatibility can to some extent be allowed.

The EDXML SDK contains reference implementations of both relative and absolute validators.

9.2.3 Collision Resolution

EDXML compliant data stores must implement collision resolution. As explained in section 5.2.5, an input event that has the same sticky hash as an existing event in the storage system is said to collide with the existing event. Collisions can be resolved by merging the events. The merge must be carried out according to the value of the `merge` attribute of each of the event properties and may change the original event. This is the only exception to the transparancy rule mentioned earlier.

9.3 Processing Systems

Processing systems that edit EDXML streams (filtering events, modifying events, ...) must preserve the per event type order of events as they are received on input. This requirement follows from the need to have order sensitive merge operations work correctly (the `replace` merge strategy) and from the fact that two events having a different event types will never merge, as they have different sticky hashes. Note that this requirement still allows EDXML processing systems to use event type specific output buffers, which may help to reduce the number of event groups in the output EDXML stream.

9.4 Infinite EDXML Streams

EDXML compliant systems may or may not support 'infinite' EDXML streams. Infinite EDXML streams are typically used in situations where EDXML systems are required to continuously provide near realtime updates of data from one or more dynamic information systems. The data stream may continue to flow indefinitely, which means that in memory processing is not possible. Note that due to its structure, chopping up EDXML streams into multiple size limited EDXML streams is possible. This means that systems that perform in memory processing can be mixed with systems that support incremental, streaming data processing.

The EDXML SDK fully supports implementing streaming EDXML processing solutions as well as incremental generation and validation of EDXML streams.

10. EDXML Design Recommendations

The importance of careful design of EDXML streams cannot be overstated. EDXML streams do not only contain data, but also hints used for knowledge inference, visualization and reporting. Therefore, EDXML enabled information processing systems rely heavily on the quality and correctness of EDXML streams. Different EDXML designs can lead to vastly different analysis results. This section will provide some important recommendations to be taken into consideration when designing EDXML data.

10.1 EDXML Stream Compatibility

Only compatible EDXML data streams can be merged. When your EDXML data is generated by a variety of sources, it is very important to make sure that the generated EDXML streams are mutually compatible. Systems that bump into inconsistent event types or object type definitions have no means to determine which definition is the right one. Consistency is most critical for object type definitions, as these are typically shared by multiple different sources. All sources that generate EDXML data containing the same object type must generate *exactly* the same object type definition. An object type library needs to be established that all EDXML generators adhere to.

The EDXML SDK contains tooling which can help you to discover, pinpoint and resolve consistency issues.

10.2 Naming of event types, properties and object types

Depending on your dataset, analysts may be faced with large collections of event types, object types and properties. It is therefore recommended to incorporate a hierarchical naming scheme. This allows analysts to quickly find the event type or object type they are looking for. To this end, the specification allows the dash '-' to be used in both event type, property and object type names. As an example, we provide a set of Internet related object type names:

- `internet-asn`
- `internet-netname`
- `internet-host-ipv4`
- `internet-host-ipv6`
- `internet-host-name`

10.3 Property descriptions

Property descriptions should always be kept short, just enough to clarify the role of the objects of the property within the context of the event type. A description like "sender" will work better than "email address of the sender of the email message". Applications can combine the event type and object type descriptions with the property description to generate a more elaborate description of a property when needed.

10.4 Property relations

Property relations can be used by analysis systems to construct networks of related events. A property relation always indicates a relation between objects within a single event.

The relation reporter strings are used to construct descriptions of relations in plain English. For this reason, the reporter strings should be constructed to complete a sentence like

Objects A and B are related because ...

For example, using a reporter string like

```
[[emailmessage-sender]] sent an email to [[emailmessage-
    recipient]]
```

in combination with a relation predicate of 'communicates with', machines can now generate texts like:

alice@mail.com communicates with bob@mail.com, because alice@mail.com sent an email to bob@mail.com.

Intra relations do not necessarily need to connect one entity defining property to another. For example, one might want to relate an account number to a client's name. Using intra relations, this enables systems to construct an entity profile by hopping from an email address to a clients account number in event A, use the account number to hop from event A to event B, and use event B to hop from the account number to the client's name. When the entity profile is presented to the user, any properties that are not entity-defining - like an account number - can be filtered out.

10.5 Use of the `replace' merge strategy

It must be noted that when your EDXML implementation makes use of the `replace` event merging strategy, the order of loading EDXML files into analysis systems generally makes a difference. Therefore, you should make sure that the systems and or persons responsible for feeding EDXML files into analysis systems take this into account.

A special use case for the replace strategy is implementing event types featuring optional properties, which can be set by updating existing events that lack these optional properties. Suppose that you have an information system that produces EDXML data streams that are stored in a large EDXML database. By default, the information system does not produce objects for all event properties, because objects of some special properties require lots of processing resources to produce. Now suppose that part of the dataset in your information system turns out to be particularly interesting, and you would like to update these events in the database such that these specific events *do* have all optional objects, not just the default objects. You can do that by setting the merge strategy of the special, optional properties to `replace`. Now a repeated upload of events to the database, now containing objects of the optional properties, will automatically update the existing events in the database, adding the objects only if the objects were not yet present. Note that in this specific use case, merge order is not an issue.

10.6 Use of floating point data types

In case you make use of hashing to refer to specific events, there is a catch with respect to the use of floating point data types. Due to the intrinsic numerical instability of floating point numbers, the hash values of events may become unstable as well. For this reason, the EDXML hashing method ignores floating point object values in the hash computation. The consequence is that events which have only floating point object values can no longer be uniquely identified by their hash value. Therefore, it is not recommended to define event types which have only floating point object values.

11. The concept of Entity

As you may have noticed, the EDXML specification contains several entity related attributes, but entities cannot be explicitly defined. Rather, they 'emerge' from the data. The rationale behind this is that entities are not always easy to identify and discriminate. When you use EDXML to represent client records from a database, defining entities representing these clients is really quite straight forward. However, when you need to study a large collection of documents, these documents may contain hundreds of references to the name 'alice' without a clear indication if all of these references are in fact the same person or not. Of course, one could define one entity for every occurrence of a name in the document collection. This will yield hundreds of entities named 'alice', each which may or may not in fact be the same person as another entity.

Especially in eDiscovery and forensic science, constructing a collection of well defined, clearly distinguishable entities can be a very difficult problem. EDXML does not solve this problem, but acknowledges the complexity of the concept of entity and provides means to deal with this complexity in a relatively simple and elegant way. Rather than allowing entities to be explicitly defined, the ontology defines hints describing how entities can be discovered in the dataset. This process of entity discovery involves connecting objects together using intra and inter relations. This yields constructs of related objects which together can be regarded as entities. These entities may be clearly distinguishable from one another, or they may be very fuzzy, all depending on the confidences specified in properties and property relations.

Entities really are products of an analysis process rather than static, a priori constructed units of information. They can even become dynamic when live streaming data sources are used, changing as more information becomes available over time.

Note that an entity does not need to represent a person. It can be anything that cannot be identified using any single identifier. However, in EDXML 2.1 it is not possible to discriminate between different types of entity. This will probably change in EDXML 3.0.

12. Future Developments

EDXML v2.1 is the last feature update in the 2.x series. This series is showing its age and has acquired some inconsistencies and oddities that have been kept in place for the sake of backward compatibility. The next EDXML version will be EDXML 3.0, which will break compatibility with the 2.x series. In general, EDXML 3.0 will be an opportunity to fix issues in the current 2.x series to arrive at a specification that is cleaner, simpler, more elegant, more consistent and more in line with common practise.

Some of the more specific changes that are being considered for EDXML 3.0 are listed below.

sticky hashes	The algorithmn for computing sticky hashes will change to include the source URL.
versioning	Event type definitions will have a `version` attribute. This attribute will be used to update event type definitions on live running systems. The requirement that EDXML data stores should only accept event type definitions that are identical to previously consumed definitions can now be broken in a controlled way.
translation	The `translation` tag is considered to be too specific to be included in the EDXML specification itself. Translations are just one of many possible forms of information that one may want to associate with EDXML events, varying from tags and annotations to arbitrary files. In order to keep the specification clean, this tag will have to go. Associating arbitrary forms of information with EDXML evenst can still be achieved using sticky hashes.
optional attributes	Some of the attributes that are currently optional will become mandatory, in order to simplify EDXML implementations.
entity types	It will be possible to discriminate between different types of entity.
timestamps	The `timestamp` data type will be changed into an ISO 8601 timestamp. This evades known limitations with some XML processing software, and extends the range of dates which can be represented.
object elements	The `object` elements that contain object values will be wrapped in a new `property` element, which groups object values that share the same property. This will allow generation of XML schemas for specific event types and allow event data validation to be delegated to standard XML validators.
security	Features will be added to allow both the authenticity and integrity of EDXML data to be verified.

13. Tag and Attribute Reference

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

Tag	Parent	Attribute	Description
`events`			Root tag
`definitions`	`events`		Contains all definitions.
`eventtype`	`definitions`		Event type definition.
		`name`	Name of event type.
		`display-name`	Display name of event type.
		`description`	Description of event type.
		`classlist`	Comma separated list of event type classes.
		`reporter-short`	Reporter string (short version).
		`reporter-long`	Reporter string (full version).
`parent`	`eventtype`		Optional parent definition.
		`eventtype`	Event type of parent.
		`propertymap`	Property mapping from child to parent.
		`parent-description`	Describes relation between child and parent.
		`siblings-description`	Describes relation between child and siblings.
`properties`	`eventtype`		Contains event type properties.
`property`	`properties`		Property definition.
		`name`	Name of property.
		`description`	Short description of property.
		`object-type`	Name of the associated object type.
		`unique`	Optional property uniqueness ('true' or 'false'). Default is 'false'.
		`similar`	Optional hint for finding similar events.
		`merge`	Optional merge strategy, defaults to 'drop'.
		`defines-entity`	Optional flag for entity identifying properties ('true' or 'false'). Default is 'false'.
		`entity-confidence`	Optional floating point value between 0.0 and 1.0. Default is zero.
`relations`	`eventtype`		Contains property relations.
`relation`	`relations`		Relation definition.
		`property1`	Property name.
		`property2`	Property name.
		`description`	Description of relation.
		`type`	Type of relation, including predicate.
		`confidence`	Floating point value between 0.0 and 1.0.
		`directed`	Optional directedness indicator ('true' or 'false'). Default is 'true'

Tag	Parent	Attribute	Description
`objecttypes`	`definitions`		Contains all object types.
`objecttype`	`objecttypes`		Object type definition.
		`name`	Name of object type.
		`display-name`	Display name of event type.
		`description`	Description of object type.
		`data-type`	An EDXML data type.
		`compress`	Optional compression hint ('true' or 'false'). Default is 'false'.
		`enp`	Optional Entity Naming Priority ([0,255]). Defaults to zero.
		`regexp`	Optional regular expression. Defaults to `[\s\S]*`.
		`fuzzy-matching`	Optional fuzzy comparison option.
`sources`	`definitions`		Contains all sources.
`source`	`sources`		Source definition.
		`source-id`	Source identifier.
		`url`	URL representation of source.
		`description`	Description of source.
		`date-acquired`	Acquisition date in yyyymmdd format.
`eventgroups`	`events`		Contains all event groups.
`eventgroup`	`eventgroups`		Event group.
		`event-type`	Name of event type.
		`source-id`	Source identifier.
`event`	`eventgroup`		Event.
		`parents`	Optional list of parent hashes.
`object`	`event`		Object definition.
		`property`	Name of the property of the object.
		`value`	Object value.
`content`	`event`		Optional plain text event content.
`translation`	`event`		Optional content translation (deprecated).
		`language`	ISO 639-1 language code of the optional content translation.
		`interpreter`	Identifier of the interpreter.