UNIVERSITE DE GENEVE

CENTRE UNIVERSITAIRE
D'INFORMATIQUE
GROUPE VISION

Date:
N° 99.04

First version: October 6, 1999
This version: June 22, 2000

TECHNICAL REPORT

VISION

MRML: Towards an extensible standard for multimedia querying and benchmarking

Draft Proposal

Wolfgang Müller Zoran Pečenović[1] Arjen P. de Vries[2*]
David McG. Squire Henning Müller Thierry Pun

Computer Vision Group
Computing Science Center, University of Geneva
24 rue du Général Dufour, CH - 1211 Geneva 4, Switzerland

[1]Laboratoire de Communications Audio-Visuelles and Groupe de l'Ergonomie des Systèmes Intelligents
in the Institut de Microtechnique (IMT) Ecole Polytechnique Fédérale de Lausanne, Switzerland

[2]Department of Computer Science
University of Twente, The Netherlands

e-mail: Wolfgang.Mueller@cui.unige.ch,
Zoran.Pecenovic@lcav.de.epfl.ch

*Arjen was happy to test that thanks still worked.

Abstract

In this technical report we introduce and describe the Multimedia Retrieval Markup Language (MRML). This XML-based markup language is the basis for an open communication protocol for content-based image retrieval systems (CBIRSs). MRML was initially designed as a means of separating CBIR engines from their user interfaces. It is, however, also extensible as the basis for standardized performance evaluation procedures.

Such a tool is essential for the formulation and implementation of common benchmarks for CBIR. A common protocol can also bring new dynamics to the CBIR field—it makes the development of new systems faster and more efficient, and opens the door of the CBIR research field to other disciplines such as Human-Computer Interaction. The MRML specifications, as well as the first MRML-compliant applications, are freely available and are introduced in this technical report.

Keywords: multimedia database,interoperability,query language

1 Introduction

The development of research can be described as moving in two directions

- search for new, useful query and interaction paradigms
- deeper research to improve the performance of systems that have adopted a given query paradigm.

The search for new better performance given a query paradigm has led to "clusters" of systems which are similar in their interaction with the user, and which give a certain set of interaction capabilities to the user.

It is already visible that research will move towards systems which enable the user to formulate multi–paradigm queries in order to further improve results.

As a consequence of the above, there is the need for

- A common mechanism for shipping multi–paradigm queries and their results. Such a mechanism assures that the right query processor processes the right query.
- For each paradigm a common language, which allows to formulate queries in this paradigm.

Fulfilling these needs would enable the different communities to share user interfaces. At present almost every group has its own interface suiting its purposes. However, many interfaces are very much alike. Fulfilling these needs would also enable the different communities to share implementations without sharing the code. Comparing the resulting systems of different groups of a research domain would be easier, if one could make them accessible to outside scripts without publishing the actual code.

The query shipping–mechanism should be designed in a way that it does not constrain ongoing and future research regarding both the search for optimal query formulation for each paradigm and the research for new query paradigms for MMDB queries. In short, the query–shipping mechanism should *separate the communication problem from the query formulation problem*, letting research groups evolve freely to find good domain specific query formulation schemes.

MRML, as proposed in this technical report, provides such a shipping–mechanism. In addition to the requirements stated above it was designed for making the use of it as

simple as possible, thus allowing groups with exotic development environments and little manpower to be able to use MRML.

MRML in the current version provides a complete solution for QBE in CBIRS, which was the initial use of MRML.

MRML is an XML based language for the communication between MMDB server and user interface. In this technical report we demonstrate the utility of the framework provided by MRML by giving the example how we use MRML in our *Viper* system.

The technical report is organized as follows:
First we describe the general framework with emphasis on extensibility. After this we give the first application of MRML which is the use in our CBIRS systems together with the CIRCUS interface written by one of us (Z.P.).

2 The design of the MRML query shipping framework

Common positions always limit the freedom of the individual. However, in this case the design is easily extensible. In this section we propose a development strategy which will preserve the freedom of the individual research groups, while keeping the standard.

When designing MRML we were lead by the following goals:

Extensibility Our main concern was to provide a framework which permits independent growth of the products of different research groups (followed by periodical code merging).

No preferred implementation language We want to leave the developer the freedom of choice of the implementation language. A standard like this is unlikely to be adopted by the research community, if it works only with a given "mainstream" computing environment.

Independence of third–party libraries We want the use of the communication protocol to be as independent from third party libraries as possible. A group should be able to provide its own tools within finite time.

Our choice is to use an XML (eXtensible Markup Language) DTD (Document Type Definition – a grammar) for the specification of our communication protocol, together with specifications for the transmission of messages, and for extensions of the protocol.

When making this choice we saw mainly the alternative of using EJB (Enterprise Java Beans), CORBA, and other methods of remote procedure calls. However, we feared strong links with languages (Java/EJB) or large program packages (CORBA). A further advantage of XML is that the use of an XML for communication directly implies a common *human readable* log file format of this communication.

The attractiveness of XML is further increased by the existence of free tools in numerous programming languages. XML has been designed explicitly for simplifying parser design. XML has to be parsable by deterministic parsers, thus it is simple to implement one's own XML/MRML parsers.

2.1 The structure of XML and "graceful degradation"

The structure of XML is similar to that of HTML, which stems from their common ancestry, *i.e.* SGML (Standard Generalized Markup Language): an XML document can be seen as a tree of "elements" which themselves contain other elements. The content of each node of the document tree is a list of attribute-value pairs, as well as a sequence of nodes

(possibly interleaved with text). This structure is encoded using so called "tags" for the elements. The "opening tag" of an element with type `t` and attribute `anAttribute` being set to `x` would be <`t anAttribute="x"`>. The "closing tag" of an element `t` would be < /`t`>.

This free structure is constrained by a Document Type Definition (DTD) which is a grammar for the tree structure. The details can be found in [1].

Graceful degradation is the key to successful independent extension of MRML. The basic principles can be summarized as follows:

- servers and clients which do not recognize an XML element or attribute encountered in an MRML text should completely ignore its contents,
- extensions should be designed such that all the standard information remains available to the generic MRML user (see examples in § 3). In other words, the modifications of the standard MRML should be as local as possible therefore not create global changes which, from the above would be dismissed by standard parsers (see the examples in section 3).

These principles provide guidelines for independent extensions of MRML. To avoid conflicts between differing extensions of MRML, we plan to maintain or promote a central database for the registration and documentation of MRML extensions. This would also facilitate the "translation" between user logs which contain extended MRML.

2.2 A walk through of MRML

MRML-based communications have the structure of a remote procedure call: the client connects to the server, sends a request, and stays connected to the server until the server breaks the connection. The server shuts down the connection after sending the MRML message which answers the request. This connectionless protocol has the advantage of easing the implementation of the server. To limit the performance loss caused by frequently reconnecting, it is possible to send several requests as part of a single MRML message. The extension of MRML to a protocol permitting the negotiation of a permanent connection is also planned.

MRML, in its current specification (and implementation) state, supports the following features:

- request of a capability description from the server,
- selection of a data collection classified by query paradigm; it is possible to request collections which can be queried in a certain manner,
- selection and configuration of a query processor, also classified by query paradigm; MRML also permits the configuration of meta-queries during run time,
- formulation of QBE queries,
- transmission of user interaction data.

The final feature reflects our strong belief that affective computing [6] will soon play a role in the field of content-based multimedia retrieval. MRML already supports this by allowing the logging of some user interaction data. In particular, this is the case for the history-forward and history-backward functionalities of the SnakeCharmer interface.

2.3 Logging onto a CBIR server

An MRML server listens on a port for MRML messages on a given TCP socket. When connecting, the client requests the basic properties of the server, and waits for an answer. Skipping standard XML headers, the MRML code looks like this:

```
<mrml>
 <get-server-properties />
</mrml>
```

The server then informs the client of its capabilities. This message is empty in the current version of MRML, but it allows for the extension of the protocol:

```
<mrml>
 <server-properties />
</mrml>
```

Using similar simple messages, the client can request a list of the collections available on the server, together with descriptions of the ways in which they can be queried.

The client can then open a session on the server, and configure it according to the needs of its user (interactive client) or its own needs (*e.g.* meta-query agents). The client can also request the algorithms which can be used with a given collection:

```
<mrml>
 <get-algorithms
  collection-id="collection-1" />
</mrml>
```

This request is answered by sending the corresponding list of algorithms. This handshaking mechanism allows both interactive clients and programs (such as meta-query agents or automatic benchmarkers) to obtain information describing the server.

In a similar simple manner, the client can open and close sessions for a user, and configure the algorithms chosen by the user. This enables multi-user servers and also on-the-fly learning by the query processor.

2.4 Interface configuration

The client can then request property sheet descriptions from the server. Different algorithms will have different relevant parameters which should be user-configurable (*e.g.* feature sets, speed vs. quality). *Viper*, for example, offers several weighting functions [7] and a variety of methods for, and levels of, pruning. All these parameters are irrelevant for CIRCUS. Thanks to MRML property sheets, the interface can adapt itself to these specific parameters. At the same time, MRML specifies the way the interface will turn these data into XML to send them back to the server. Here is short example of interface configuration:

```
<property-sheet
 property-sheet-id="s1"

 type="numeric"
 numeric-from="1"
 numeric-to="100"
 numeric-step="1"

 caption="\% features evaluated"

 send-type="attribute"
 send-name="cui-percentage-features" />
```

This specifies a display element which will allow the user to enter an attribute with the caption "% of features evaluated". The values the user will be able to enter are integers between 1 and 100 inclusive. The value will be sent as an attribute *e.g.* `cui-percentage-features="33"`. This mechanism allows the use of complex property sheets, which can send XML text containing multiple elements. The interested reader is referred to the appendix for details.

2.5 Query Formulation

The query step is dependent on the query paradigms offered by the interface and the search engine. MRML currently includes only QBE, but it has been designed to be extensible to other paradigms.

A basic QBE query consists of a list of images and the corresponding relevance levels assigned to them by the user. In the following example, the user has marked two images, the image `1.jpg` positive (`user-relevance="1"`) and the image `2.jpg` negative (`user-relevance="-1"`). All query images are referred to by their URLs.

```
<mrml session-id="1" transaction-id="44">
<query-step session-id="1"
 resultsize="30"
 algorithm-id="algorithm-default">
 <user-relevance-list>
  <user-relevance-element
   image-location="http://viper.unige.ch/1.jpg"
   user-relevance="1"/>
  <user-relevance-element
   image-location="http://viper.unige.ch/2.jpg"
   user-relevance="-1"/>
 </user-relevance-list>
</query-step>
</mrml>
```

The server will then return the retrieval result as a list of images, again represented by their URLs.

Queries can be grouped into transactions. This allows the formulation and logging of complex queries. This may be applied in systems which process a single query using a variety algorithms, such as the split-screen version of *TrackingViper* [4] or the system described by Lee *et al.* [3]. It is important in these cases to preserve in the logs the knowledge that two queries are logically related one to another.

3 Extending MRML

In order to demonstrate how easily MRML can be extended to other query paradigms, we give as an example QBE for images with user annotation. We assume that the user is invited to associate textual comments with images he or she marks as relevant or irrelevant. Since a tag for this purpose does not yet exist in MRML, we add an attribute `cui-user-annotation="..."` to the element. The prefix `cui-` is added to avoid name clashes with extensions from other groups which use MRML.

```
<user-relevance-list>
 <user-relevance-element
  image-location="file:/images/1.jpg"
  user-relevance="1"
  cui-user-annotation="tropical fish"/>
</user-relevance-list>
```

It is important to note here that servers which do not recognize the `cui-user-annotation` attribute still can make use of the remaining information contained in the `user-relevance-element` element.

As an example of how *not* to extend MRML, we give an extension with the same semantics but which does not respect the principle of graceful degradation:

```
<user-relevance-list>
 <cui-user-relevance-element
  image-location="file:/images/1.jpg"
  user-relevance="1"
  user-annotation="tropical fish">
</user-relevance-list>
```

Instead of adding an *attribute* to an existing MRML element (`user-relevance-element`), a new *element* was defined that contained the same kind of extension, namely `cui-user-relevance-element`. Consequently, servers which do not recognize this element will not be able to exploit any relevance information.

4 Future work in the specification of MRML

There are two main directions concerning further work on MRML and related goals.

1. Enhancing MRML: it is already clear at the time of writing, that MRML as is, very flexible, already very useful, but incomplete. We need to incorporate (at least):

   - region queries
   - text in queries

   Here we are hoping for cooperation with working groups who are using these query techniques in their systems. We would like to make MRML a "living standard", always keeping language specification and implementation date close together.

   An analysis of what could be done in the future can be found in [2].

2. Providing tools: In our opinion the best way of using the advantages created by MRML is to pool common tools which can be used and exchanged within the research community.

5 State and future of the implementation

NOT implemented are

- the `multiset` property sheet element,
- the enforcement of `subset` size constraints
- the the `visibility` attribute (everything is `popup`)
- the client side of enabling meta algorithms

Everything else is implemented demonstrated under: `http://viper.unige.ch/`.

References

[1] Extensible markup language (xml) 1.0 (w3c recommendation 10-february-1998).

[2] Yuan-Chi Chang, Lawrence Bergmann, John R. Smith, and Chung-Sheng Li. Query taxonomy of multimedia databases. In Panchanathan et al. [5]. (SPIE Symposium on Voice, Video and Data Communications).

[3] Catherine S. Lee, Wei-Ying Ma, and HongJiang Zhang. Information Embedding Based on User's Relevance Feedback for Image Retrieval. In Panchanathan et al. [5]. (SPIE Symposium on Voice, Video and Data Communications).

[4] Wolfgang Müller, David McG. Squire, Henning Müller, and Thierry Pun. Hunting moving targets: an extension to Bayesian methods in multimedia databases. In Panchanathan et al. [5]. (SPIE Symposium on Voice, Video and Data Communications).

[5] Sethuraman Panchanathan, Shih-Fu Chang, and C.-C. Jay Kuo, editors. *Multimedia Storage and Archiving Systems IV (VV02)*, volume 3846, Boston, Massachusetts, USA, September 20–22 1999. (SPIE Symposium on Voice, Video and Data Communications).

[6] Rosalind W. Picard. *Affective Computing.* MIT Press, Cambridge, 1997.

[7] Gerard Salton and Chris Buckley. Term weighting approaches in automatic text retrieval. Technical Report 87-881, Department of Computer Science, Cornell University, Ithaca, New York 14853-7501, November 1987.

A Further documentation of MRML

A.1 State machine of MRML client–server communication

Client–server–communication in MRML is a sequence of connections. In each connection a single request or a small groups of requests is answered by the server using a single message or a small group of messages. The state machine in fig. A.1 describes the communication starting with the point where the first makes contact with the server.

The client establishes first contact with the server by sending a `get-server-properties` message. As a response the client receives a `server-properties` which is empty for standard MRML. However, this message is an important stub for extensions which concern the connection itself (e.g. finding out, if the server is able to do a session using a permanent connection.

After receiving the configuration description, the client will ask for a list of sessions for a user, using the `get-sessions` tag. The reply is a `session-list`. The client will now open one session using `open-session`, getting an `acknowledge-session-op` as return. The opened session is required to have a sensible default state, *i.e.* a state which allows queries.

Opening the session, the server has received the user's name, password and the session-id. *I.e.* it has all the necessary information for knowing which collections and algorithms the user should see. Please note, that no one is forced to do user-dependent configuration of the system, but MRML gives the possibility of doing so. So, after opening the session, the client has the possibility to request both lists of collections and algorithms.

Both collections and algorithms are described by the query-paradigms they allow, as well as some other parameters. In particular, an algorithm can contain as an attribute the ID of a collection on which it will be used.

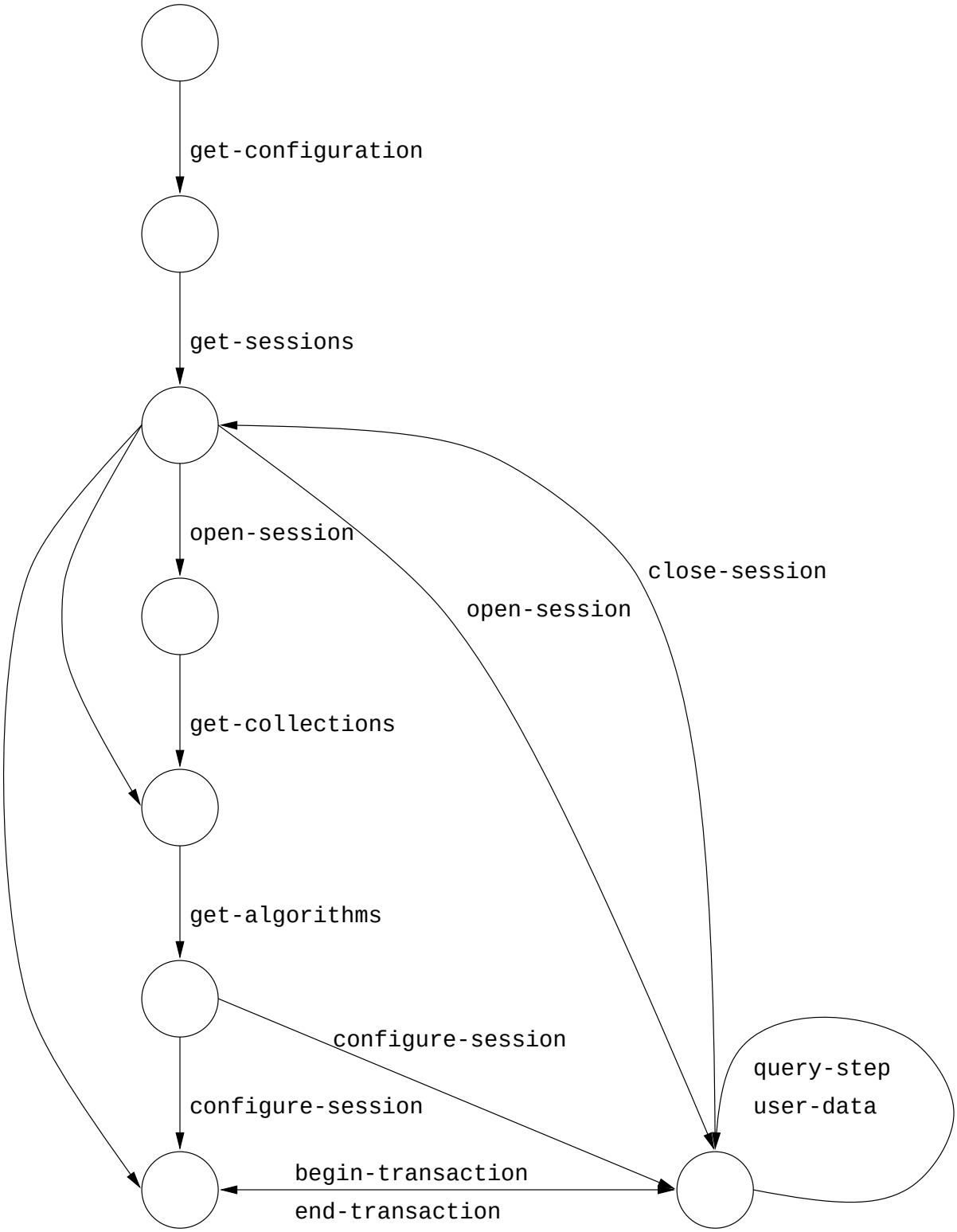

Figure 1: *The state grammar of MRML client–server communication.*

Getting both a list of collections and a list of algorithms, the client has enough information to configure the session which has been opened: when configuring the session, the client sends a `configure-session` signal which contains an `algorithm` with the attributes `algorithm-id` and `algorithm-type` set. The attribute `collection-id` a suitable algorithm.

After this, the session is fully configured and can be queried (using `query-step`). Queries can be grouped into transactions for group queries for logging and learning purposes.

Intermixed with the queries, the client is able to send `user-data` to the server. These user data tags contain user interaction information for logging and learning purposes.

A.2 `query-paradigms`

Algorithms are described by their `algorithm-id` and their `algorithm-type`, as well as by their `query-paradigm-list`. A `query-paradigm-list` contains `query-paradigm` elements which contain an unspecified number of attributes. One of which can be the attribute `query-mode` which at present has the possible values "qbe" or "browsing". All other attributes presently are extensions.

The main use of the query paradigm list is to enable clients to determine which collection can be used with which algorithm. In short, an algorithm can used with a collection, if their `query-paradigm-lists` *match*.

Two `query-paradigm-lists` l_1 and l_2 match, if there is at least one pair of `query-paradigms` $e_1 \in l_1$, $e_2 \in l_2$ such that e_1 and e_2 match. Two `query-paradigms` $e_{1,2}$ match, if for the sets of their attribute–value pairs $S_{1,2}$ holds:

$$((a, v_1) \in S_1 \textbf{ and } (a, v_2) \in S_2) \implies (v_1 = v_2)$$

In particular, a `query-paradigm` tag without attributes matches any other `query-paradigm` tag.

A.3 Algorithms

As it was said, Algorithms are described by their `algorithm-id` and their `algorithm-type`, as well as by their `query-paradigm-list`, and (optionally) a `allows-children` element (which in turn contains another `query-paradigm-list`).

As described in the last section, the first `query-paradigm-list` specifies which collection can be queried with this algorithm, *and* it informs the client about its properties. The client or its user can then decide if to proceed or not.

It is possible to specify algorithms recursively. Algorithms can contain other algorithms, possibly several of one type `algorithm-type`, the `algorithm-id` however, has to be unique in one `configure-session` statement. It is thus possible to let the client specify meta-queries. Which kind of meta queries can be built, decided by the `allows-children` tag. An algorithm a_1 is allowed to contain another algorithm a_2, if the `query-paradigm-list` contained in the `allows-children` tag of a_1 matches the `query-paradigm-list` of a_2.

A.4 MRML property sheets

MRML property sheets are a method to work around the fact that the a common set of configuration parameters for image databases is difficult to find and probably awkward

to use. We suggest to achieve this by sending code which allows to build GUIs (*i.e.* the subset you would need for configuration of an algorithm), along with a specification of how to generate pieces of XML code from the GUI's state. This code is XML and it will not be executed, so, to our knowledge, there is no inherent security hole.

A.4.1 A simple example

Viper is a system which uses inverted files for the indexation of images. Each image is translated in a variable–length sequence of features which describe the image. Each feature is assigned a weight determined dependent on the frequency of the feature within the image and within the collection. How exactly this is done, depends on the weighting functions. Both retrieval performance and processing speed of the system depend on the weighting function.

Viper gives the possibility to choose the weighting function at runtime, using an attribute `cui-weighting-function` of the `algorithm` element. The following property sheet gives the possibility to choose between two weighting functions.

The "basic need" of a system would be to specify the collection, *i.e.* the database on which the retrieval is to be performed. For testing and comparison it would be interesting to have the choice between several algorithms (e.g. wavelet coefficient/color histogram based).

A choice out of a list of two elements:

```
<property id="p1"
          type="subset"
          caption="Weighting function"
          visibility="visible"

          sendtype="attribute"

          sendname="cui-weighting-function"

          minsubsetsize="1"
          maxsubsetsize="1">

          <property id="p2"
                    type="setelement"
                    caption="Best fully weighted"
                    visibility="visible"

                    sendtype="value"

    sendvalue="best-fully";


                    defaultstate="selected"
          >
          </property>
          <property id="p3"
                    type="setelement"
                    caption="Classical IDF"
```

```
                    visibility="visible"

                    sendtype="value"

    sendvalue="classical-idf";

                    defaultstate="unselected"
          </property>
</property>
```

What does this do exactly?

- it defines a list of which the user is allowed to chose a `subset` of size between `1` and `1`, *i.e.* an exclusive choice.
- When asked for its state this list will generate an `attribute`, *i.e.* the text given by `send-name`, plus `=: cui-weighting-function=`.
- The value of the attribute will be determined as follows: follows: `property` Elements `p1` and `p2` are identical in structure. They denote the elements of our set which can either be `selected` or `unselected`. *If selected* they send a text which will be placed like an attribute value (`value`). This text will be `"best-fully"` or `"classical-idf"`, depending on which of the two list items is chosen by the user.

As a result: the piece of MRML above will enable the interface to set up a property sheet which comprises a list of two items, of which one can be selected. Depending on the selection, the interface will send either

```
cui-weighting-function="best-fully"
```

or

```
cui-weighting-function="classical-idf"
```

to the server. The MRML client will use this property sheet when generating a `configure-session` message.

A.4.2 More complex: generate XML subtrees

The following example describes the generation of whole document subtrees. This feature is not yet immediately useful for *Viper* or CIRCUS. However it provides an explanation on how the text generated in the previous is included into `configure-session` message. More important is the fact that it provides a general framework for describing (GUI) entities which can send XML.

Consider the following example: Imagine an algorithm which runs the query image through a series of filters before running them through a simple query processor. Being a research system, we would like these filters to be run-time-configurable. Each filter needing some parameters, the and number of filters being variable, we simply need to define some new MRML tags which permit us to describe the sequence of filters. We would an output like the one given below:

```
<cui-filter-list>
    <cui-filter cui-filter-type="horizontal-gabor"
                cui-filter-gabor-sdev="50"
                cui-filter-gabor-wavelength="10"/>
    <cui-filter cui-filter-type="gauss" cui-filter-gauss-sdev="5"/>
</cui-filter-list>
```

The corresponding property sheet would look like:

```
<property id="p1"
          type="panel"
          caption="Filter Sequence"
          visibility="invisible"

          send-type="element"

          send-name="cui-filter-sequence"
>
    <property id="p1"
              type="multi-set"
          caption="Filter"
          visibility="visible"

          send-type="element"

          send-name="cui-filter"

          minsubsetsize="0"
          maxsubsetsize="5">

          <property id="p11"
                    type="set-element"
                    caption="Gaussian blur"
                    visibility="pop-up"

                    sendtype="attribute"
                    sendname="cui-filter-type"
                    sendvalue="gauss"

                    defaultstate="selected"
          >
                    <property id="p111"
                              type="numeric"
                              caption="Standard deviation"
                              visibility="pop-up"

                              sendtype="attribute"
                              sendname="cui-filter-gauss-sdev"

                    />
```

```
        </property>
        <property id="p12"
                  type="set-element"
                  caption="Horizontal Gabor"
                  visibility="pop-up"

                  sendtype="attribute"
                  sendname="cui-filter-type"
                  sendvalue="horizontal-gabor"

                  defaultstate="selected"
        >
                  <property id="p121"
                            type="numeric"
                            caption="Tile size"
                            visibility="pop-up"

                            sendtype="attribute"
                            sendname="cui-filter-gabor-sdev"

            numeric-from="5"
      numeric-to="100"
      numeric-step="5"
                  />
                  <property id="p122"
                            type="numeric"
                            caption="Tile size"
                            visibility="pop-up"

                            sendtype="attribute"
                            sendname="cui-filter-gabor-wavelength"

            numeric-from="2"
      numeric-to="20"
      numeric-step="1"
                  />
        </property>
    </property>
</property>
```

The example above shows exactly the described scenario: The user has the choice to use sequences of 0 up to 5 filters. The filters can be either Gaussian blur or horizontal gabor filters (yes, this is a toy example).

The Gaussian blur can be configured by giving a number between 1 and 100, which will be sent as an attribute (`cui-filter-gauss-sdev`). The gabor filter can be configured using the two parameters `cui-filter-gabor-sdev` and `cui-filter-gabor-wavelength`.

Both the configuration panels will `pop-up` when the corresponding filter has been selected in the sequence.

In the following section we describe how the text is actually generated, and how dialog

dynamics is specified.

A.4.3 A more formal description of MRML property sheets

As it has become clear from the examples, GUIs sent using MRML property sheets are in fact a tree of property sheets. Both the XML generated by the property sheet and the dialog dynamics are defined using simple rules.

Dialog dynamics A `property` element is *visible on the screen*, if

1. all its ancestors are visible
2. AND
   - its parent is non–selectable OR selected
   - OR its parent has the `visibility="visible"` attribute set.

selectability of `property` elements will be defined below.
A `property` element is `active`, if

1. all its ancestors are active
2. AND its parent is non–selectable OR selected

An active `property` element is defined as an element that can be *used for its purpose*, *i.e.* it will be enabled on the GUI screen.

Generating XML XML is generated during a depth-first-traversal of the `property` tree as follows:

- The XML string generated by a sequence of active elements is equal to the concatenation of the XML strings generated by each element. The sequence of concatenation is equal to the physical sequence of `property` elements in the MRML text.
- The XML string generated by an inactive element is empty.
- The XML string generated by an active element is given by the `send-type` of the `property` element

  `send-type="element"`: If there is any beginning of an opening tag in the XML generated by the ancestors of this `property` element, it will be ended by adding an `>` to the text generated so far.

  Afterwards this `property` element will generate the beginning of the opening tag of an XML element with a name that is specified by the attribute `sendname`, followed by a space and the content of the attribute `send-value`. As an example: if for a given element the `sendname` attribute has the value `xxx`, and the content of the `send-value` attribute is `'myattribute="5"'`, the generated output will be `<xxx␣myattribute="5"`.

  After that the children are evaluated in sequence and a closing tag of the element will be generated. Before that, the opening tag will be ended, if necessary. (`/</xxx>/`, in our example)

`send-type="attribute"`: If there is no beginning of an opening tag in the XML generated by the ancestors of this property sheet no text will be generated.

If there is any beginning of an opening tag in the XML generated by the ancestors of this property sheet there are the following possibilities:

`value` **is nonempty** Generate the text given by the values of the attributes `sendname` and `send-value` in the definition of this `property`. For example `sendname="myattribute" send-value="33"` will lead to the text `myattribute="33"` as output.

`value` **is empty** begin an attribute definition with a name given by the value of the attribute `sendname`. For example `sendname="myattribute" send-value=""` will lead to the text `myattribute=` as output. The actual definition of the value can be provided in two ways:

- If the current `property` has an inherent value (*i.e.* is `numeric`, `boolean` or `textual`), this value is taken, and thus the attribute definition will be ended.
- The value definition will be provided by a child.

`send-type="value"`: If there is no attribute definition, which has been begun by any ancestor or sibling of this `property` element, no text is generated.

Otherwise either the inherent value or the value given by the attribute value `send-value` will be used, as described above.

`send-type="none"`: This `property` element will not generate any code.

B The DTD of MRML

Here is the documented DTD of MRML. It has been derived from the DTD used by *Viper* via a Perl script. We removed all attributes and tags which are *Viper*–specific extensions to MRML and by added some highlighting of the comments:

```
<!--
```

Basic structure: Messages are sent as MRML texts. In order to make it easy for the server to know who connects, each message is assigned the id of its session as an attribute.

Author of this file: Wolfgang Mueller with lots of suggestions and corrections from
David Squire, Arjen P. de Vries and Christoph Giess

```
-->

<!ELEMENT mrml (begin-transaction?,(

                get-configuration
                |configuration-description

                |get-sessions
                |session-list

                |open-session
```

```
                |rename-session
                |close-session
                |delete-session

                |get-collections
                |collection-list

                |get-algorithms
                |algorithm-list

                |get-property-sheet
                |property-sheet

                |configure-session

                |query-step
                |query-result

                |user-data

                |error

                )?,
               end-transaction?)
>

<!ATTLIST mrml
                 session-id ID #IMPLIED
                 transaction-id ID #IMPLIED
>

<!--
```

Request: *get-configuration*

This is the message an MRML client sends to the server on connection. The message `get-configuration` *gives information about the basic server configuration.*

```
-->

<!ELEMENT get-configuration EMPTY
>

<!--
```

Response: *configuration-description*

The `get-configuration` *message is anwered by a message which is supposed to hold a description about anything which is nonstandard MRML.*

```
-->
```

```
<!ELEMENT configuration-description EMPTY
>

<!--
```

Request: *get-sessions*

The `get-sessions` *message permits the user to request exsisting sessions for a given user. It is sent by the client directly after after the* `configuration-description` *has been delivered, and prior to any other activity.*

Authentification happens before any other activity to give the server the possibility to customise any other information sent to the user. For example, it might be sensible to send different property sheets to different classes of users, or to render some parts of the database only visible to the own work group.

```
-->

<!ELEMENT get-sessions EMPTY
>

<!ATTLIST get-sessions
                        user-name CDATA #REQUIRED
                        password CDATA "guest"
>

<!--
```

Response: *session-list*

A session is described by its `session-id`*. We also send over a more human-readable name*

```
-->

<!ELEMENT session-list (session+)
>

<!ELEMENT session EMPTY
>

<!ATTLIST session
                    session-id CDATA #REQUIRED
                    session-name CDATA "Default session"
>

<!--
```

Request: *get-collections*

gets a list of collections on the server.

```
-->
```

```
<!ELEMENT get-collections EMPTY
>

<!--
```

Request: *collection-list*

a list of collections on the server.
a collection is described by a list of the of the query paradigms which can be used for querying it, as well as an ID and its human-readable name.
This means, you do not have to index all collections using all the methods you want to propose to the server.

```
-->

<!ELEMENT collection-list (collection*)
>

<!ELEMENT collection (query-paradigm-list?)
>

<!ATTLIST collection
                        collection-id CDATA #REQUIRED
                        collection-name CDATA #REQUIRED
>

<!--
```

Tag: *query-paradigm*
arises both in algorithms and collections: this describes the kind of query which can be performed with this algorithm/collection

```
-->

<!ELEMENT query-paradigm-list (query-paradigm*)
>

<!ELEMENT query-paradigm EMPTY
>

<!ATTLIST query-paradigm
                            query-paradigm-id CDATA #REQUIRED
>

<!--
```

Request: *get-algorithms*

gets a list of algorithms usable for one collection

```
-->
```

```
<!ELEMENT get-algorithms EMPTY
>

<!ATTLIST get-algorithms
                          collection-id CDATA #IMPLIED
                          query-paradigm-id CDATA #IMPLIED
>

<!--
```

Response: *algorithm-list*

returns a list of algorithms for a given collection on the server

```
-->

<!ELEMENT algorithm-list (algorithm*)
>

<!--
```

Tag: *algorithm*

An algorithm can contain other algorithms, optionally a property sheet, optionally a query paradigm list optionally an "allows-children" specification

```
-->

<!ELEMENT algorithm (algorithm*,property-sheet?,query-paradigm-list?,allows-children?)
>

<!ATTLIST algorithm
                      algorithm-id CDATA #REQUIRED
                      collectiom-id CDATA #REQUIRED
                      algorithm-name CDATA #REQUIRED
                      algorithm-id ID #REQUIRED
                      algorithm-type CDATA "adefault"
                      collection-id CDATA "cdefault"
>

<!--
```

Tag: *allows-children*

This tag specifies for an algorithm what kind of algorithms can be children of this algorithm. no specification $\Rightarrow$ allows no children.

```
-->

<!ELEMENT allows-children (query-paradigm-list?)
>

<!--
```

Request: *get-property-sheet*
get the property sheet for an algorithm

```
-->

<!ELEMENT get-property-sheet EMPTY
>

<!ATTLIST get-property-sheet
                               algorithm-id ID #REQUIRED
>

<!--
```

Request: *begin-transaction*
begins a transaction

```
-->

<!ATTLIST begin-transaction
                             transaction-id ID #REQUIRED
>

<!--
```

Request: *end-transaction*
ends a transaction

```
-->

<!ATTLIST end-transaction
                           transaction-id ID #REQUIRED
>

<!--
```

Request: *configure-session*
configures the session by giving an algorithm

```
-->

<!ELEMENT configure-session (algorithm)
>

<!--
```

Tag: *property-sheet*

Basic idea: *send a property sheet together with a specification what should be the XML output coming back. Useful for configuring your database.*

```
-->

<!ELEMENT property-sheet (property-sheet)*
>

<!ATTLIST property-sheet
                          property-sheet-id ID #REQUIRED
                          property-sheet-type (
                                               multi-set

                                               |subset
                                               |set-element
                                               |boolean
                                               |numeric
                                               |textual
                                               |panel

                                               |clone
                                               |reference) #REQUIRED
                          caption CDATA #IMPLIED
                          visibility (popup|visible|invisible) "visible"
                          send-type (element
                              | attribute
                              | attribute-name
                              | attribute-value
                              | children
                              | none) #REQUIRED
                          send-name CDATA #IMPLIED
                          send-value CDATA #IMPLIED
                          min-subset-size CDATA #IMPLIED
                          max-subset-size CDATA #IMPLIED
                          from CDATA #IMPLIED
                          step CDATA #IMPLIED
                          to CDATA #IMPLIED
                          auto-id (yes|no) #IMPLIED
                          auto-id-name CDATA "id"
                          defaultstate CDATA #IMPLIED
>

<!--
```

Tag: *algorithm*

An algorithm will be either an empty element with attributes (add some at your will, it will talk with your server anyway, and this is the server which sent the property sheet), or a tree of algorithms.

What is the use of this? Think of configuring meta queries. Together with properties you get a powerful tool.

```
-->
```

```
<!ELEMENT algorithm (algorithm*)
>

<!--
```

Beginning and renaming sessions

We want to give the user the possibility to save the current state into "sessions". This might be useful in the case that a user has several classes of goals which s/he knows in advance.

The user can request a new session. S/he can also request a name change for a session.

Ending sessions is implicit: we cannot afford being dependent on the user ending his/her session in a "nice" way, so we do not tempt programmers to do so

```
-->

<!--
```

Interface side

```
-->

<!--
```

send the desired feedback method together with a name for the session

```
-->

<!ELEMENT open-session EMPTY
>

<!ATTLIST open-session
                        user-name CDATA #REQUIRED
                        password CDATA #IMPLIED
                        session-id CDATA #IMPLIED
                        session-name CDATA #IMPLIED
>

<!ELEMENT rename-session EMPTY
>

<!ATTLIST rename-session
                          session-id CDATA #IMPLIED
                          session-name CDATA #IMPLIED
>

<!ELEMENT delete-session EMPTY
>

<!ATTLIST delete-session
                          session-id CDATA #REQUIRED
>
```

```
<!ELEMENT close-session EMPTY
>

<!ATTLIST close-session
                          session-id CDATA #REQUIRED
>

<!ELEMENT acknowledge-session-op EMPTY
>

<!ATTLIST acknowledge-session-op
                                    session-id CDATA #REQUIRED
>

<!--
```

Simple user commands (for logging purposes)
(like e.g. back or forward in the command history) (at present the only commands)

```
-->

<!ELEMENT user-data EMPTY
>

<!ATTLIST user-data
                      command (history-backward|history-forward) "backward"
                      steps CDATA #IMPLIED
>

<!--
```

Request: *query-step*

At present we provide only query by example, and search for random images (done, if one sends an empty `query-step` *tag)*

```
-->

<!ELEMENT query-step (user-relevance-element-list?)
>

<!ATTLIST query-step
                       query-step-id CDATA #REQUIRED
                       result-size CDATA #IMPLIED
                       result-cutoff CDATA #IMPLIED
                       query-type (query|at-random) "query"
                       algorithm-id CDATA #IMPLIED
>

<!--
```

Tag: *user-relevance-element-list*

List of URLs with user given relevances Our way of specifying a QBE for images. relevances vary from 0 to 1

```
-->

<!ELEMENT user-relevance-element-list (user-relevance-element+)
>

<!ELEMENT user-relevance-element EMPTY
>

<!ATTLIST user-relevance-element
                                  user-relevance CDATA #REQUIRED
                                  image-location CDATA #REQUIRED
>

<!--
```

Response: *query-result*

each result image can be accompanied by the user given relevance, as well as the similarity calculated by the program, based on the feature space.
calculated similarities vary from 0 to 1

```
-->

<!ELEMENT query-result (query-resultelement-list?,query-result*)
>

<!ELEMENT query-result-element-list (query-resultelement+)
>

<!ELEMENT query-result-element EMPTY
>

<!ATTLIST query-result-element
                                 calculated-similarity CDATA #REQUIRED
                                 thumbnail-location CDATA #REQUIRED
                                 image-location CDATA #REQUIRED
>

<!--
```

Error messages.

```
-->

<!ELEMENT error EMPTY
>

<!ATTLIST error
                 message CDATA #REQUIRED
>
```