Overview of ACE XML 2.0

ACE XML is a set of file formats that are designed to meet the special representational needs of research in music information retrieval (MIR) research in general, and automatic music classification research in particular. The ACE XML formats are designed to represent a wide range of musical information clearly and simply using formally structured frameworks that are flexible and extensible. ACE XML 2.0 represents a fundamental redesign of the earlier ACE XML 1.1 standard.

Summarized as briefly as possible, the overarching design priorities behind ACE XML are the maximization of expressivity, flexibility and extensibility while at the same time maintaining as much simplicity and accessibility as possible. These two sets of often opposed design priorities are addressed by making many of the ACE XML elements and attributes optional so that they can be included or omitted as needed. This makes it possible to use simple and concise files when this is all that is necessary, and to only necessitate the use of those particular additional options that are needed when more expressivity is required.

As implied by their name, ACE XML files are all XML-based. XML was chosen because it is not only a standardized format for which parsers are widely available, but is also extremely flexible while at the same time able to clearly specify data structuring that can be as rigid as needed. XML is verbose, with the consequence that it is less space efficient than formats such as ARFF, but this verbosity has the compensating advantages of increasing human readability and of allowing greater flexibility and extensibility.

There are five types of ACE XML files that perform the following core purposes:

- *Feature Value:* These files express feature values extracted from specific instances.
- *Feature Description:* These files express abstract information about features themselves (but not extracted feature values).
- *Instance Label:* These files associate class labels with specific instances and their subsections. They can be used to express either ground truth labels or predicted labels output by a classification system.
- *Class Ontology:* These files express relationships between classes. They can be used to simply catalogue candidate class labels, or for more sophisticated purposes such as expressing weighted ontological relationships between classes.
- *Project:* These files indicate associations between related groups of ACE XML files and other resources.

This is only a brief outline of the expressivity offered by each of the file types, however, as is evident from the individual descriptions of each of them in the sections below. Of particular note, all of the ACE XML file types offer functionality for specifying a variety of useful metadata.

Each of these XML file types may be used independently, or they may be associated with one another and integrated in software using unique identifiers. It is not in any way necessary to provide each of the ACE XML file types for any application if this is not appropriate or needed.

For example, if a classifier is already trained and is to be used to classify unknown patterns, then there is no need for an Instance Label file, although one may be produced during processing to express predicted classes.

The ACE XML code base, discussed briefly below, automatically constructs implied data for missing file types in a way that is hidden from the user. For example, if only a Feature Value file and an Instance Label file are specified, the software can automatically construct a flat class ontology based on the labels present in the Instance Label file, and can also automatically generate feature descriptions based on the characteristics of the features present in the Feature Value file, such as the dimensionality of each feature.

The decision to use multiple different file types rather than the more typical single file type is unorthodox, and requires some justification. It is useful to incorporate a separation between feature values and instance labels. This is important for data reusability, such as in a case where one might extract features once from a large number of recordings, and then reuse the single resulting Feature Value file for multiple purposes, such as classification by performer, composer, genre and mood. If there were only one ACE XML file type, then feature values would need to be repeated for each of these applications. With the ACE XML approach, however, the Feature Value file can remain unchanged and be reused with a different Instance Label file for each classification task. Similarly, one can imagine a case where the same model classifications contained in one Instance Label file are used for separate sets of features extracted from symbolic, cultural and audio data contained in three different Feature Value files.

Feature descriptions and class ontologies are each distributed in separate files as well in order to emphasize their independence from particular instances. For example, a Feature Description file could be published on its own to catalogue the features that can be extracted by a particular feature extraction application in general, or to express specific extraction parameters that were found to be effective for a particular research domain. Similarly, class ontologies could be published in a way that is independent of particular instances and features, or even of a particular dataset as a whole.

Such file type separations emphasize the abstract nature of many of the types of data that are useful in music classification, and allow the files to be distributed and used either independently or together, as appropriate, rather than artificially forcing connections where they may not always be appropriate. The use of separate file formats also has advantages with respect to data longevity and convenience when updating data. If new features become available after a Feature Value file has been generated, for example, it would only be necessary to update the Feature Value and Feature Description files since the data stored in the other two file types could be reused unmodified. Similarly, if a class ontology changed over time, it would not be necessary to update Feature Value or Feature Description files.

Overall, and most importantly, the separation of different types of data into different file types makes it possible to distribute and use one type of file for arbitrary purposes without needing to impose particular choices with respect to the types of data described by the other file types. The separation into multiple files types also makes it easier to conceptualize and represent sophisticated arrangements of information with a divide and conquer approach.

It was decided to use XML DTDs rather than some other XML schema to define the ACE XML file types. Although other schemas can in general be more expressive than DTDs, DTDs are nonetheless sufficiently expressive for the purposes of ACE XML. They also have the advantages of being simpler and easier to understand, thereby making the ACE XML formats more attractive to new users. This is a particular advantage considering the variety of alternative schemas that are available. The average member of the MIR community is less likely to be familiar with any particular one of these alternative schema languages, particularly in the cases of those specialized schemas that provide enough increased expressivity to have advantages over simple DTDs. Furthermore, DTDs tend to be easier to learn for those users who might not know any XML at all. The ACE XML DTDs are specified in each ACE XML file along with the file's data, which means that each ACE XML file is packaged with an explanation of its data structuring.

Incorporating ACE XML 2.0 into Existing and New Software

A key factor in the effectiveness of any effort to encourage music researchers to adopt new standardized file formats is the ease with which they can parse and write to them in their own existing and new software. The ample data structures and processing functionality offered by the Weka code base have certainly contributed to its broad adoption, for example.

Specialized code libraries are thus being implemented for public release in June 2009 to support the ACE XML file formats. These include functionality for parsing, writing and merging ACE XML files as well as performing various utility functions such as batch annotating files into an Instance Label file, generating Instance Label files based on simple tab delimited text files or accessing data from iTunes XML files. These libraries also include data structures for storing ACE XML data that can be accessed by external code via simple and well-documented APIs. This data can be used and manipulated directly using these libraries, or it can be exported to individual developers' own data structures.

These code libraries are open source and are distributed for free with a GNU GPL. They are implemented in Java in order to ensure portability, and do not rely on any technologies beyond Java that require any kind of special installations. The resulting portability is important in helping to encourage adoption by less technically oriented users.

Functionality has also been implemented to automatically convert data stored in ACE XML data structures into Weka data structures, and vice versa, in order to take advantage of the convenient and well-established functionality built into Weka. This also makes it possible to use Weka data structures as intermediaries for conversion to yet other formats. jMIR also includes utilities for directly translating back and forth between Weka ARFF and ACE XML files, although data that fundamentally cannot be represented in ARFF files is lost when doing so.

An ACE XML GUI editor is also implemented as part of the ACE software. This makes it possible to conveniently view, edit and save ACE XML files. Of course, ACE XML files can also be edited using simple text editors or specialized software like Altova XML Spy. The jMIR applications also include functionality for writing and, when appropriate, reading ACE XML files.

Linking ACE XML 2.0 Files to External Resources

Each of the ACE XML file formats allow unlimited numbers of optional *uri* clauses to be added in a wide variety of contexts. Each *uri* element has an optional *predicate* attribute in order to make it possible to specify RDF-like triples, such that the file context where the *uri* element is found indicates the subject, the contents of the *uri* element specifies the object, and the *predicate* attribute specifies the relationship between the two. So, for example, in the case of an Instance Label ACE XML file providing information about genre labels, the following *uri* statement could be placed in a *class* clause associated with Afro-Cuban jazz in order to provide a link to useful background information:

```
<uri predicate="http://en.wikipedia.org/wiki/Music_genre">
http://www.allmusic.com/cg/amg.dll?p=amg&sql=77:2601</uri>
```

Such *uri* elements provide a powerful and flexible way of linking ACE XML files to external resources. This can be used simply to provide links to references, or it can be used to provide hooks to formally integrate ACE XML files into frameworks like RDF ontologies

uri elements are by no means obligatory in ACE XML files, however, and it is perfectly acceptable to omit them if desired. They are not in fact used in any core ACE XML or jMIR functionality, so ACE XML files can still be used in an entirely self-contained way even if *uri* tags are present. In other words, the *uri* tags are provided only to facilitate integration with external resources or into external frameworks, but they in no way introduce dependencies that limit the usability of ACE XML files if these resources become unavailable, as can be the case with RDF ontologies, and they do not compromise the backwards compatibility of future updates to ACE XML

In the special case of related ACE XML files, each of the ACE XML file formats offers the option of using a *related_resources* clause. Such a clause can be used to reference one or more related ACE XML Feature Value, Feature Description, Instance Label, Class Ontology or Project files, respectively referenced via *feature_value_file, feature_description_file, instance_label_file, class_ontology_file* or *project_file* elements. Although related ACE XML files can also be grouped using an ACE XML Project file, the *related_resources* element provides the option of forming more general connections with a scope broader than just a particular project. *uri* tags may also optionally be used inside a *related_resources* clauses to reference as many external resources as desired that are relevant to the document as a whole.

ACE XML 2.0 Feature Value Files

Feature Value files are used to express feature values that have been extracted for instances as a whole, for subsection regions of instances or at quantized coordinates in an instance. This can be useful, for example, when dealing with windowed audio feature extractions or with symbolic feature extractions where features are extracted separately for different sections of a score. The Feature Value DTD is shown in Figure 1.

```
<!ELEMENT ace_xml_feature_value_file_2_0 (comments?, related_resources?,
                                          instance+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT related_resources (feature_value_file*,
                             feature_description_file*,
                             instance_label_file*,
                             class_ontology_file*,
                             project_file*,
                             uri*)>
<!ELEMENT feature_value_file (#PCDATA)>
<!ELEMENT feature_description_file (#PCDATA)>
<!ELEMENT instance_label_file (#PCDATA)>
<!ELEMENT class_ontology_file (#PCDATA)>
<!ELEMENT project_file (#PCDATA)>
<!ELEMENT uri (#PCDATA)>
<!ATTLIST uri predicate CDATA #IMPLIED>
<!ELEMENT instance (instance_id,
                    uri*,
                    extractor*,
                    coord_units?,
                    s*,
                    precise_coord*,
                    f*)>
<!ELEMENT instance_id (#PCDATA)>
<!ELEMENT extractor (#PCDATA)>
<!ATTLIST extractor fname CDATA #REQUIRED>
<!ELEMENT coord_units (#PCDATA)>
<!ELEMENT s (uri*,
             f+)>
<!ATTLIST s b CDATA #REQUIRED
            e CDATA #REQUIRED>
<!ELEMENT precise_coord (uri*,
                         f+)>
<!ATTLIST precise_coord coord CDATA #REQUIRED>
<!ELEMENT f (fid,
             uri*,
             (v+ | vd+ | vs+ | vj))>
<!ATTLIST f type (int | double | float | complex | string) #IMPLIED>
<!ELEMENT fid (#PCDATA)>
<!ELEMENT v (#PCDATA)>
<!ELEMENT vd (#PCDATA)>
<!ATTLIST vd d0 CDATA #REQUIRED d1 CDATA #IMPLIED d2 CDATA #IMPLIED
             d3 CDATA #IMPLIED d4 CDATA #IMPLIED d5 CDATA #IMPLIED
             d6 CDATA #IMPLIED d7 CDATA #IMPLIED d8 CDATA #IMPLIED
             d9 CDATA #IMPLIED>
<!ELEMENT vs (d+,
              v)>
<!ELEMENT d (#PCDATA)>
<!ELEMENT vj (#PCDATA)>
```

Figure 1: The XML DTD for the ACE XML 2.0 Feature Value format.

The fundamental units of Feature Value files are *instance* clauses. A number of different kinds of metadata may be expressed for an instance by including the following elements within its *instance* clause:

- *instance_id:* The contents of this obligatory element specify a unique identifier for an instance. This may be a network URI, a file path, a publication number, a title, a hash code, or any other type of identifying string that is convenient, as long as it is uniquely used with reference to the given instance. *instance_id* tags are also used to link extracted feature values with corresponding class labels stored in ACE XML Instance Label files.
- *uri:* Described in the section above.
- *extractor:* This optional element may be used to specify the name of the feature extraction software that was used to extract features. A separate *extractor* clause is needed for each feature. The contents of an *extractor* clause indicate the name of the feature extractor, and the obligatory *fname* attribute indicates the name of the feature that is to be associated with this extractor. This arrangement accommodates scenarios where different feature extractors are used to extract the same feature for different instances, as well as scenarios where different features are extracted by different feature extractors for the same instance.
- *coord_units:* This optional element may be used to specify units for the coordinate values associated with the *b, e* and *coord* attributes described below.

Actual feature values and instance subdivisions are specified after this metadata in *instance* clauses. In the case of subsections, the feature values are wrapped in an *s* clause that includes obligatory *b* and *e* attributes to specify the beginning and end coordinates of the section, respectively. There may be multiple subsections within an instance that may or may not overlap, that may or may not be of equal size and that may or may not comprehensively cover the overall instance. This makes it possible to have, for example, overlapping analysis windows of arbitrary and potentially varying sizes. There is nothing about the *b* and *e* attributes that specifically associates them with time, although they often are in practice. They could just as easily be used to specify a range of pixels in an image of album art, for example.

In the case of a subsection that corresponds to a quantized point (e.g., a single audio sample or a single pixel), with only a single corresponding coordinate, a *precise_coord* clause is used instead of an *s* clause, with a single *coord* attribute specifying the point within the instance that is referred to. Both *s* and *precise_coord* clauses may make use of optional *uri* tags to refer to external resources.

Each instance or subsection may contain an arbitrary and potentially differing number of features in an arbitrary and potentially differing order. This makes it possible to omit features from some instances or subsections if appropriate or if they are unavailable.

The value(s) of a feature are specified within an *f* clause. A separate *f* clause is used for each feature. The *f* element may be specified within an *instance* clause at the root level or within a subsection *s* clause or a *precise_coord* clause. The following information may be specified within each *f* clause:

- *fid:* The contents of this obligatory element specify a unique identifier for a feature. This identifier may be any string, as long as it is only used in reference to the feature in question. *f_id* tags may be used to link Feature Value files to ACE XML Feature Description files in order to access more information about individual features if desired.
- *type:* This optional attribute may be used to specify the data type that should be used to store the feature value. The options are integers, doubles, floats, complex numbers and text strings. Although this typing is not strictly necessary for jMIR, it is sometimes necessary for other software. *type* is assumed to be double by the jMIR ACE XML parser if it is not specified in an *f* clause, but this not an intrinsic assumption of the Feature Value format. Feature types may also be specified in an associated Feature Description file, in which case the they can be omitted from the Feature Value file for the sake of brevity.
- *v, vd, vs* or *vj:* Feature values themselves are enclosed in any one of these elements. These are each described below.

Before proceeding to explain the differences between these four feature value tags, it is useful to first provide several examples. Figure 2 demonstrates a case where only features extracted for an instance overall are encoded, Figure 3 adds features extracted over subsections of the instance, and Figure 4 adds a variety of metadata. These examples illustrate how ACE XML files can be kept relatively sparse and simple if appropriate, but can also be very expressive if needed.

```
<instance>
   <instance_id>C:\Audio\song_57.mp3</instance_id>

   <f>
      <fid>Duration</fid>
      <v>257</v>
   </f>
   <f>
      <fid>Bit Rate</fid>
      <v>160</v>
   </f>
</instance>
```

Figure 2: An excerpt from an artificial ACE XMLE 2.0 Feature Value file indicating feature values extracted for a single instance that, in this case, is an MP3 file. This example specifies two single-value features extracted from the recording as a whole. The identifier for the instance is a file path in this case, but it could just as well be a URI, a fingerprint string or any other unique key.

```
<instance>
   <instance_id>C:\Audio\song_57.mp3</instance_id>

   <s b="0" e="150">
      <f>
         <fid>Spectral Centroid</fid>
         <v>1016.8</v>
      </f>
      <f>
         <fid>RMS</fid>
         <v>0.1559</v>
      </f>
   </s>
   <s b="125" e="257">
      <f>
         <fid>Spectral Centroid</fid>
         <v>980.5</v>
      </f>
      <f>
         <fid>RMS</fid>
         <v>0.1229</v>
      </f>
   </s>

   <f>
      <fid>Duration</fid>
      <v>257</v>
   </f>
   <f>
      <fid>Bit Rate</fid>
      <v>160</v>
   </f>
</instance>
```

Figure 3: An excerpt from an artificial ACE XMLE 2.0 Feature Value file indicating feature values extracted for a single instance. This example expands on Figure 2 by adding features extracted over two overlapping analysis windows. Two features are extracted for each window.

```
<instance>
   <instance_id>C:\Audio\song_57.mp3</instance_id>
   <uri predicate="http://www.mpeg.org/">
      http://audio.datastte.org/files/song_57.rdf</uri>
   <extractor fname="jAudio">Spectral Centroid</extractor>
   <extractor fname="jAudio">RMS</extractor>
   <extractor fname="jAudio">Duration</extractor>
   <extractor fname="Audacity">Encoding</extractor>
   <coord_units>ms</coord_units>

   <s b="0" e="150">
      <f type="double">
         <fid>Spectral Centroid</fid>
         <v>1016.8</v>
      </f>
      <f type="double">
         <fid>RMS</fid>
         <v>0.1559</v>
      </f>
   </s>
   <s b="125" e="257">
      <f type="double">
         <fid>Spectral Centroid</fid>
         <v>980.5</v>
      </f>
      <f type="double">
         <fid>RMS</fid>
         <v>0.1229</v>
      </f>
   </s>

   <f type="int">
      <fid>Duration</fid>
      <v>257</v>
   </f>
   <f type="int">
      <fid>Bit Rate</fid>
      <v>160</v>
   </f>
</instance>
```

Figure 4: An excerpt from an artificial ACE XML 2.0 Feature Value file indicating feature values extracted for a single instance. This figure builds upon Figure 3 by adding several kinds of metadata, including a link to an imagined RDF ontology associated with the instance, the name of the feature extraction software used to extract each feature, the units used to denote subsection coordinates and the data type of each feature.

One essential requirement for a file format designed to express feature values is the ability to represent feature arrays of arbitrary dimensionalities, not just single value features or feature vectors. Also, some features may have a dimensionality that varies from extraction to extraction, so it should be possible to accommodate this. Another essential consideration is efficiency of representation, since files can quickly become very large if there are many features extracted over many subsections of many instances, particularly when features have high dimensionalities. It should also be possible to specify sparse arrays, or arrays that are missing some elements. Finally, one would ideally like feature values to be relatively human readable for debugging purposes, although this is not the most important consideration.

As mentioned above, there are four different elements that may be used to represent feature values in Feature Value files. Each of these have different strengths and weaknesses with respect to the above considerations, and are each suitable for different kinds of features. An efficient file writer may use different encodings in the same file in order to minimize file size, so each *f* clause may use any one of the four feature encoding tags, independently of what is used in any other *f* clauses. Each of the four encoding formats works as follows:

- *v:* This simple approach permits only single-value features or non-sparse feature vectors. A separate *v* element is used for each value of a feature vector, with the index of the value implied by the order in which the *v* clauses appear.
- *vj:* Arrays with any number of dimensions may be expressed using JSON array notation. JSON is a well-established and relatively human readable text-based data interchange format for representing simple data structures. JSON arrays are expressed using simple square bracket notation, which are in turn enclosed in *vj* elements in ACE XML 2.0. So, for example, a feature vector of size three consisting of the numbers one, two and three would be represented as *<vj>[1,2,3]</vj>*. JSON arrays can be nested in order to represent arrays of arbitrary dimensionality. A table with two identical rows, for example, each containing the values one, two and three would be represented as *<vj>[[1,2,3],[1,2,3]]<vj>*. A similar approach could have been achieved by using nested XML elements, but the JSON representation is standardized and more compact. There are also open source JSON libraries available in many languages that can parse JSON arrays quickly, which offloads some of the work from that would otherwise need to be performed by the ACE XML parser. A disadvantage of the JSON approach, however, is that JSON is not ideally suited to representing sparse arrays. Also, JSON can be less human readable than some of the alternative approaches.
- *vd:* This approach involves explicitly notating the coordinates of entries in feature arrays using ten attributes numbered *d0* to *d9*. If there is only one coordinate (i.e., a feature vector), then only the d0 attribute would be used, if there is a three-dimensional array then the d0, d1 and d2 attributes would be used, and so on. This approach is less space efficient than JSON, but can represent sparse arrays very well and is relatively human readable. There is a limitation to only ten dimensions, but each of these dimensions may be of any size, and very few features used in MIR need arrays with more than ten dimensions. Although it would be ideal to have such an approach for N dimensions, it is not possible to specify an arbitrary number of attributes in an XML DTD schema. To give an example, the *JSON* feature vector *[1,2,3]* would be represented as:

  ```
  <vd d0="0">1</vd><vd d0="1">2</vd><vd d0="2">3</vd>
  ```

and the JSON array *[[1,2,3],[4,5,6]]* would be represented as:

```
<vd d0="0" d1="0">1</vd><vd d0="0" d1="1">2</vd><vd d0="0" d1="2">3</vd>
<vd d0="1" d1="0">4</vd><vd d0="1" d1="1">5</vd><vd d0="1" d1="2">6</vd>
```

- *vs:* This final option may be used to represent arrays of any dimensionality, including sparse arrays, and is the only option with this degree of generality. Its disadvantage is that it is less concise than JSON and potentially less human readable than the other options. Each *vs* clause contains one *d* element for each dimension, and this *d* element is used to specify the coordinate value in its corresponding dimension. The dimension corresponding to a *d* element is implied by the order in which the *d* elements appear. Each *vs* clause also contains a single *v* clause to specify the feature value for the array at the corresponding coordinate. To give an example, the *JSON* feature array *[[1,2,3],[4,5,6]]*would be represented as:

```
<vs><d>0</d><d>0</d><v>1</v></vs>
<vs><d>0</d><d>1</d><v>2</v></vs>
<vs><d>0</d><d>2</d><v>3</v></vs>
<vs><d>1</d><d>0</d><v>4</v></vs>
<vs><d>1</d><d>1</d><v>5</v></vs>
<vs><d>1</d><d>2</d><v>6</v></vs>
```

Figure 5 demonstrates the use each of the four encoding formats.

It should be noted that all of these figures only indicate data for a single instance each. In practice, Feature Value files typically include data for multiple instances, with the consequence that they contain multiple instance clauses.

As noted above, it is possible to link feature values for instances stored in Feature Value files to class labels and other metadata associated with the same instances that is stored in ACE XML Instance Label files. Similarly, features extracted in Feature Value files can be linked to feature parameters and other general feature metadata stored in ACE XML Feature Description files. This can be done using matching *instance_id* and *fid* tags, respectively. Although this can certainly be helpful, Feature Value files can also stand on their own, as the ACE XML software libraries can automatically implicitly deduce information such as the dimensionality of features without Feature Definition files if needed. It can often be more efficient to store information such as feature data types in Feature Definition files, however.

```
<instance>
   <instance_id>Artificial Example</instance_id>
   <f>
      <fid>Single Value</fid>
      <v>1</v>
   </f>
   <f>
      <fid>1-D Vector</fid>
      <v>1</v>
      <v>2</v>
      <v>3</v>
   </f>
   <f>
      <fid>Another 1-D Vector</fid>
      <vd d0="0">1</vd>
      <vd d0="1">2</vd>
   </f>
   <f>
      <fid>2-D Table</fid>
      <vd d0="0" d1="0">1</vd>
      <vd d0="0" d1="1">2</vd>
      <vd d0="0" d1="2">3</vd>
      <vd d0="1" d1="0">11</vd>
      <vd d0="1" d1="1">22</vd>
      <vd d0="1" d1="2">33</vd>
   </f>
   <f>
      <fid>The Same 2-D Table</fid>
      <vs><d>0</d><d>0</d><v>1</v></vs>
      <vs><d>0</d><d>1</d><v>2</v></vs>
      <vs><d>0</d><d>2</d><v>3</v></vs>
      <vs><d>1</d><d>0</d><v>11</v></vs>
      <vs><d>1</d><d>1</d><v>22</v></vs>
      <vs><d>1</d><d>2</d><v>33</v></vs>
   </f>
   <f>
      <fid>3-D Array</fid>
      <vj>[[[[1],[2],[3],[4]],
            [[11],[22],[33],[44]],
            [[111],[222],[333],[444]]],
           [[[4],[5],[6],[7]],
            [[44],[55],[66],[77]],
            [[444],[555],[666],[777]]]]
      </vj>
   </f>
</instance>
```

Figure 5: An excerpt from an ACE XML 2.0 Feature Value file indicating artificial feature values for a single instance. This figure demonstrates how features of different dimensionalities can be expressed using each of the v, vd, vs *and* vj *tags.*

ACE XML 2.0 Feature Description Files

Feature Description files are used to express abstract information about features in a way that is independent of particular feature extractions. Feature Description files do not specify actual feature values or other information related to specific instances, as this information is instead specified ACE XML Feature Value files. The Feature Description DTD is shown in Figure 6.

```
<!ELEMENT ace_xml_feature_description_file_2_0 (comments?,
                                               related_resources?,
                                               global_parameter*,
                                               feature+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT related_resources (feature_value_file*,
                             feature_description_file*,
                             instance_label_file*,
                             class_ontology_file*,
                             project_file*,
                             uri*)>
<!ELEMENT feature_value_file (#PCDATA)>
<!ELEMENT feature_description_file (#PCDATA)>
<!ELEMENT instance_label_file (#PCDATA)>
<!ELEMENT class_ontology_file (#PCDATA)>
<!ELEMENT project_file (#PCDATA)>
<!ELEMENT uri (#PCDATA)>
<!ATTLIST uri predicate CDATA #IMPLIED>
<!ELEMENT feature (fid,
                  description?,
                  related_feature*,
                  uri*,
                  scope,
                  dimensionality?,
                  data_type?,
                  parameter*)>
<!ELEMENT fid (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT related_feature (fid,
                          relation_id?,
                          uri*,
                          explanation?)>
<!ELEMENT relation_id (#PCDATA)>
<!ELEMENT explanation (#PCDATA)>
<!ELEMENT scope (#PCDATA)>
<!ATTLIST scope overall (true|false) #REQUIRED
                sub_section (true|false) #REQUIRED
                precise_coord (true|false) #REQUIRED>
<!ELEMENT dimensionality (uri*,
                         size*)>
<!ATTLIST dimensionality orthogonal_dimensions CDATA #REQUIRED>
<!ELEMENT size (#PCDATA)>
<!ELEMENT data_type (#PCDATA)>
<!ATTLIST data_type type (int | double | float | complex | string) #REQUIRED>
<!ELEMENT global_parameter (parameter_id,
                           uri*,
                           description?,
                           value?)>
<!ELEMENT parameter (parameter_id,
                    uri*,
                    description?,
                    value?)>
<!ELEMENT parameter_id (#PCDATA)>
<!ELEMENT value (#PCDATA)>
```

Figure 6: The XML DTD for the ACE XML 2.0 Feature Description format.

Feature Description files can be used to accompany associated Feature Value files in order to specify feature constraints, such as data types or dimensionalities, in a way that is more efficient than encoding this information in the Feature Value files directly. Additional useful metadata about features, such as extraction parameters, can be matched to extracted feature values by using both of these file types.

There are also many possible applications for self-contained Feature Description files. Examples include a catalogues of features that can be extracted by a particular feature extraction application, or a list of features and associated parameters that have been found to be useful for a particular research application, such as instrument classification or pitch classification.

Information on each feature is expressed in a separate *feature* clause. Each such clause can include the following information about the feature:

- *fid:* The contents of this obligatory element specify a unique identifier for the feature. This may be a simple feature name, a URI, or any other type of unique identifying string. *fid* tags are used to link feature information with corresponding extracted feature values stored in ACE XML Feature Value files.
- *description:* This optional element may be used to provide qualitative information about the feature.
- *related_feature:* This optional element can be used to specify other features that are related in some way to the feature under consideration. This could be used, for example, to note that one feature is an alternative implementation of another. Each *related_feature* clause contains an *fid* sub-clause identifying the related feature, an optional *relation_id* element that can be used to specify the nature of the relationship, optional *uri* tags to link to external resources and an optional *explanation* element to provide a qualitative description of the relationship between the features.
- *uri*
- *scope:* The parameters of this obligatory element indicate whether the feature can be extracted for an instance as a whole, for subsections of an instance or at quantized points in an instance.
- *dimensionality:* The *orthogonal_dimensions* attribute of this optional element can be used to constrain the number of orthogonal dimensions that values for this feature have (e.g., 1 for a feature vector, 2 for a table structure, etc.). The dimensionality clause can contain a separate *size* element for each such dimension that specifies the size of each of each corresponding dimension (e.g., the length of a feature vector, the number of rows and the number of columns in a table structure, etc.). *uri* clauses may also be used to link to external information. The dimensionality element should be omitted for features that vary in dimensionality, in which case the ACE XML parsing software will automatically dynamically vary the dimensionality of stored features as appropriate.
- *data_type:* The *type* parameter of this optional element makes it possible to specify whether the values of the feature are integers, doubles, floats, complex numbers, or strings. This information may also be specified directly in ACE XML Feature Value files, but Feature Description files take precedence in the case of contradictions, and represent the information more efficiently. Although data typing is not necessary for the jMIR components, it is sometimes necessary for other applications.

- *parameter:* Specific parameters may be associated with a feature, such as the roll-off point for the Spectral Roll-off feature, for example. Such parameters can significantly impact extracted feature values, so it is important that they be specified so that future feature extractions will be consistent. A separate *parameter* clause may be specified for each such parameter of a feature, and each such clause contains a *parameter_id* element specifying a unique identifier for the feature parameter, optional *uri* links for the parameter, an optional qualitative *description* of the parameter and the numerical (or other) *value* of the parameter.

It is also possible to specify global parameters for all features in a Feature Description file using the *global_parameter* element. This is useful for specifying overall pre-processing of files before features are extracted, for example, such as down sampling or amplitude normalization. The mechanics of *global_parameter* clauses are the same as for the *parameter* clauses.

Figure 7 provides an example of how information for a single feature can be described in a Feature Description file. In practice, Feature Description files typically contain multiple *feature* clauses, not just one.

```
<feature>
   <fid>Beat Histogram</fid>
   <description>Tempo histogram calculated using autocorrelation</description>
   <related_feature>
      <fid>Tempo Peak</fid>
      <relation_id>derivative feature</relation_id>
   </related_feature>
   <uri predicate="reference">http://www.music-ir.org/evaluation/mirex-
                                results/articles/tempo/tzanetakis.pdf</uri>
   <scope overall="true" sub_section="false" precise_coord="false"></scope>
   <dimensionality orthogonal_dimensions="1">
      <size>161</size>
   </dimensionality>
   <data_type type="double"></data_type>
   <parameter>
      <parameter_id>normalized</parameter_id>
      <value>true</value>
   </parameter>
</feature>
```

Figure 7: An excerpt from a sample ACE XMLE 2.0 Feature Description file indicating information about a single Beat Histogram *feature. It is noted that this feature is related to another feature called* Tempo Peak *that can be calculated from the* Beat Histogram *feature. A URL is also provided that indicates background information on beat histograms. In terms of more technical information, it is noted that this feature is configured to only be extracted for musical files as a whole, that it consists of a single vector of size 161, that feature values are stored as doubles and that the bin values are normalized.*

ACE XML 2.0 Instance Label Files

Instance Label files are used to specify class labels and miscellaneous metadata about instances and subsections of instances. These files are typically used to express ground truth class labels or predicted labels, but there are certainly other possible uses as well. The Instance Label DTD is shown in Figure 8.

Just as is with ACE XML Feature Value files, *instance* clauses are the fundamental units of Instance Label files. Each *instance* clause can include the following information about a particular instance as a whole:

- *instance_id:* The contents of this obligatory element specify a unique identifier for the instance. This may be a network URI, a file path, a publication number, a title, a hash code, or any other type of identifying string that is convenient, as long as it is uniquely used with reference to this instance. *instance_id* tags are used to link the information stored in Instance Label files to corresponding extracted feature values stored in ACE XML Feature Value files.
- *role:* This optional attribute of the *instance_id* element can be used to specify whether the instance's class label is ground truth intended for classifier training, ground truth intended for classifier testing and evaluation, or a label output by a classification system.
- *misc_info:* This optional element can be used to specify miscellaneous metadata about an instance. The metadata field is identified using the *info_id* element, and the metadata itself is put in an *info* clause. *uri* elements may also be used to link to external resources.
- *related_instance:* This optional element may be used to specify a relationship of any kind between the instance and any other instance, referred to via its *instance_id*. For example, it might be noted that one musical recording is a cover of another musical recording. The *relation_id* element can be used to identify the specific kind of relationship, and the *explanation* element can be used to provide a qualitative description of the relationship. *uri* links may also be specified.
- *uri*
- *coord_units:* This optional element may be used to specify units for the coordinate values associated with the *begin, end* and *coord* attributes described below.

```
<!ELEMENT ace_xml_instance_label_file_2_0 (comments?,
                                           related_resources?,
                                           instance+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT related_resources (feature_value_file*,
                             feature_description_file *,
                             instance_label_file*,
                             class_ontology_file*,
                             project_file*,
                             uri*)>
<!ELEMENT feature_value_file (#PCDATA)>
<!ELEMENT feature_description_file (#PCDATA)>
<!ELEMENT instance_label_file (#PCDATA)>
<!ELEMENT class_ontology_file (#PCDATA)>
<!ELEMENT project_file (#PCDATA)>
<!ELEMENT uri (#PCDATA)>
<!ATTLIST uri predicate CDATA #IMPLIED>
<!ELEMENT instance (instance_id,
                    misc_info*,
                    related_instance*,
                    uri*,
                    coord_units?,
                    section*,
                    precise_coord*,
                    class*)>
<!ATTLIST instance role (training | testing | predicted) #IMPLIED>
<!ELEMENT instance_id (#PCDATA)>
<!ELEMENT related_instance (instance_id,
                            relation_id?,
                            uri*,
                            explanation?)>
<!ELEMENT relation_id (#PCDATA)>
<!ELEMENT explanation (#PCDATA)>
<!ELEMENT misc_info (info_id,
                     uri*,
                     info)>
<!ELEMENT info_id (#PCDATA)>
<!ELEMENT info (#PCDATA)>
<!ELEMENT coord_units (#PCDATA)>
<!ELEMENT section (uri*,
                   class+)>
<!ATTLIST section begin CDATA #REQUIRED
                  end CDATA #REQUIRED>
<!ELEMENT precise_coord (uri*,
                         class+)>
<!ATTLIST precise_coord coord CDATA #REQUIRED>
<!ELEMENT class (class_id,
                 uri*)>
<!ATTLIST class weight CDATA "1">
<!ATTLIST class source_comment CDATA #IMPLIED>
<!ELEMENT class_id (#PCDATA)>
```

Figure 8: The XML DTD for the ACE XML 2.0 Instance Label format.

Each *instance* clause can also contain one of more *class* clauses specifying information about the class labels associated with the instance. Multiple classes can be specified by including multiple *class* clauses, or *class* clauses can be omitted entirely if no class labels are available. The following information may be specified within each *class* clause:

- *class_id:* This element is used to specify a unique identifier for the class. This element can be used to link to class data stored in ACE XML Class Ontology files that contain matching *class_ids*.
- *uri*
- *weight:* This attribute permits weighted class membership. So, for example, a given musical recording might be labeled with the Blues genre with a weight of 2 as well as with the Jazz genre with a weight of 1. Depending on context, this could mean either that the recording is a member of both the Blues and Jazz genres, but the influence of the former is twice that of the latter, or it could mean that a classifier is unsure whether the piece is Blues or Jazz, but believes that the former label is twice as likely as the latter. If the weight attribute is not specified for a class, then it is assigned a value of 1 by default. All weight values are proportional, so the absolute value of a weight has no meaning other than its value relative to the weights of other class labels.
- *source_comment:* This optional attribute specifies the source of the class label, such as an expert human annotator's name or the name of a piece of classification software.

Each *instance* clause can also specify information about subsections of the instance, denoted with a *section* element, as well as information about quantized points within the instance, denoted with the *precise_coord* element. Both of these elements can include *class* sub-clauses that operate exactly as *class* clauses for overall instances, as well as *uri* elements. The scope of each subsection is denoted using *begin* and *end* attributes, and the coordinates of quantized points are denoted with the *coord* attribute.

Subsections can be potentially overlapping and can be of potentially varying sizes. This arrangement permits two partially overlapping regions, where each region is labeled with a different class name, and the overlapping portion is associated with both labels. Such an occasion might occur, for example, in the ground truth for a music/applause discriminator where the applause in a live performance begins before the music ends. Such a situation could be equivalently expressed as either two sections with one label each overlapping in time or as three non-overlapping consecutive sections where the outer sections have one label each and the central section has two labels.

As noted above, it is possible to link instances to ACE XML Feature Value files in order to associate class labels with feature values, or to ACE XML Class Ontology files, in order to access information on the connections between different class labels. The former is fundamental to machine learning, and the latter enables certain powerful structured classification algorithms. Instance Label files can also be used entirely independently as well, such as, for example, as the save format used by annotation software.

Figure 9 provides an example of how information for a single instance can be described in an Instance Label file. In practice, an Instance Label file will typically contains multiple *instance* clauses, not just one.

```
<instance role="predicted">
   <instance_id>C:\Symbolic\piece_42.midi</instance_id>
   <misc_info>
      <info_id>Classification format</info_id>
      <info>
         Overall instances are classified by composer and
         sections are classified by form.
      </info>
   </misc_info>
   <coord_units>ms</coord_units>

   <section begin="0" end="85673">
      <class>
         <class_id>Sonata Exposition</class_id>
      </class>
   </section>
   <section begin="85674" end="278894">
      <class>
         <class_id>Sonata Development</class_id>
      </class>
   </section>
   <section begin="278895" end="525419">
      <class>
         <class_id>Sonata Recapitulation</class_id>
      </class>
   </section>
   <section begin="505787" end="515938">
      <class>
         <class_id>Symphonic Allegro</class_id>
      </class>
   </section>

   <class weight="3">
      <class_id>Haydn</class_id>
   </class>
   <class weight="1">
      <class_id>Mozart</class_id>
   </class>
</instance>
```

Figure 9: An excerpt from an artificial ACE XML 2.0 Instance Label file indicating class labels for a MIDI file. As indicated by the role *attribute, the labels are automatic classifier outputs. As indicated by the* misc_info *metadata, the subsections are classified by form and the overall instance is classified by composer. It can be seen that the piece has for the most part been structurally classified according to sonata form, although from the period of 505,787 ms to 515,938 the classifier is unsure whether it is a sonata recapitulation or an allegro in a symphony. The classifier is also unsure whether the piece is by Haydn or Mozart, but it believes that it is three times as likely to be by Haydn. In the context of a ground truth* role, *of course, such ambiguity would not be present.*

ACE XML 2.0 Class Ontology Files

Class Ontology files can be used to specify the candidate class labels for a particular classification domain and to specify weighted ontological relationships between classes. They do not, however, specify the labels of any actual instances, as this is the domain of Instance Label files, although they can be linked to Instance Label files. The Class Ontology DTD is shown in Figure 10.

```
<!ELEMENT ace_xml_class_ontology_file_2_0 (comments?,
                                           related_resources?,
                                           class+)>
<!ATTLIST ace_xml_class_ontology_file_2_0 weights_relative (true|false)
                                                            #REQUIRED>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT related_resources (feature_value_file*,
                             feature_description_file *,
                             instance_label_file*,
                             class_ontology_file*,
                             project_file*,
                             uri*)>
<!ELEMENT feature_value_file (#PCDATA)>
<!ELEMENT feature_description_file (#PCDATA)>
<!ELEMENT instance_label_file (#PCDATA)>
<!ELEMENT class_ontology_file (#PCDATA)>
<!ELEMENT project_file (#PCDATA)>
<!ELEMENT uri (#PCDATA)>
<!ATTLIST uri predicate CDATA #IMPLIED>
<!ELEMENT class (class_id,
                 misc_info*,
                 uri*,
                 related_class*,
                 sub_class*)>
<!ELEMENT class_id (#PCDATA)>
<!ELEMENT misc_info (info_id,
                     uri*,
                     info)>
<!ELEMENT info_id (#PCDATA)>
<!ELEMENT info (#PCDATA)>
<!ELEMENT related_class (class_id,
                         relation_id?,
                         uri*,
                         explanation?)>
<!ATTLIST related_class weight CDATA "1">
<!ELEMENT relation_id (#PCDATA)>
<!ELEMENT explanation (#PCDATA)>
<!ELEMENT sub_class (class_id,
                     relation_id?,
                     uri*,
                     explanation?)>
<!ATTLIST sub_class weight CDATA "1">
```

Figure 10: The XML DTD for the ACE XML 2.0 Class Ontology format.

The ability to specify ontological class structuring has several important benefits. From a musicological perspective, it provides a simple machine readable way of specifying a variety of musical relationships. From a machine learning perspective, it has the dual advantages of enabling the use of potentially very powerful hierarchical classification methodologies that take advantage of this structuring as well as the use of learning schemes utilizing weighted penalization, such that misclassifications during training into related classes are penalized less severely than misclassifications into unrelated classes.

Information on each class is expressed in a separate *class* clause. Each such clause can include the following information:

- *class_id:* This obligatory element is used to specify a unique identifier for the class. This element can be used to link to instances stored in ACE XML Instance Label files that are labeled with classes with matching *class_ids*.
- *uri*
- *misc_info:* This optional element may be used to specify miscellaneous metadata about a class. The metadata field name is specified with the *info_id* element, and the metadata itself is put in an *info* clause. Links to external resources may also be made with *uri* tags.
- *related_class:* This optional element is used to specify general ontological relationships between other classes. This relationship is unidirectional by default. Bidirectional relationships can be formed by declaring the reverse relationship in the other class' *related_class* clause.
- *sub_class:* This optional element makes it possible to specify explicitly hierarchical relationships between classes, as an alternative to the more general *related_class* approach. Classes referred to in *sub_class* clauses are hierarchically subordinate to the class from in which the *sub_class* clause is found. Tree-like structures can be built by having subordinate classes include other classes in their own *sub_class* clauses. There is no need to indicate parent classes in any way, as this relationship is implicit.

related_class and *sub_class* clauses may each contain the following information:

- *class_id:* The identifier of the related class.
- *weight:* This attribute indicates the strength of the relationship between the classes. If the weight attribute is not specified for a class, it is assigned a value of 1 by default. Weights can be either absolute or relative, as defined by the global *weights_relative* attribute, which must be either *true* or *false*.
- *relation_id:* An optional attribute identifying the kind of relationship between the classes.
- *explanation:* An optional qualitative explanation of the inter-class relationship.
- *uri*

Figure 11 demonstrates how both hierarchical and general ontological class relationships can be specified, how both unidirectional and bidirectional links can be declared, and how relationship weights can be used.

```xml
<class>
   <class_id>Robert Johnson</class_id>
</class>

<class>
   <class_id>Muddy Waters</class_id>
   <related_class weight="10">
      <class_id>Robert Johnson</class_id>
      <relation_id>Influenced By</relation_id>
   </related_class>
   <related_class weight="1">
      <class_id>Eric Clapton</class_id>
      <relation_id>Influenced By</relation_id>
   </related_class>
</class>

<class>
   <class_id>Eric Clapton</class_id>
   <related_class weight="30">
      <class_id>Robert Johnson</class_id>
      <relation_id>Influenced By</relation_id>
   </related_class>
   <related_class weight="10">
      <class_id>Muddy Waters</class_id>
      <relation_id>Influenced By</relation_id>
   </related_class>
</class>

<class>
   <class_id>Cream</class_id>
   <sub_class>
      <class_id>Eric Clapton</class_id>
      <relation_id>Had As Member</relation_id>
   </sub_class>
</class>
```

Figure 11: An excerpt from an artificial ACE XML 2.0 Class Ontology file indicating class labels consisting of names of Blues musicians and bands. Two kinds of relationships between classes are specified, namely musicians influenced by other musicians and musicians who were members of groups. In this example there is no relationship from Robert Johnson to the other musicians because he was not influenced by them. Both of the other musicians are influenced by Johnson, however. Clapton is more influenced by Johnson than by Muddy Waters, and Muddy Waters is slightly influenced by Clapton, as indicated by the weight *values. There is also a hierarchical relationship specified between Cream and Clapton, since Clapton was a member of Cream.*

ACE XML 2.0 Project Files and ZIP Files

The advantages of having four different core ACE XML formats have been made clear above. However, users in practice will often want to use multiple ACE XML files together. This can be done using *related_resources* clauses in each component file, but it would be clearer and more convenient to have an external way of specifying all of the files that are associated with any one project or application, particularly if this can be done in a way that does not compromise the reusability of the component files for entirely different projects.

The ACE XML Project file format makes this possible. This format allows users of an application such as one of the jMIR components to simply specify a single Project file, and then rely on the application to itself automatically open all of the files referred to by the Project file and update them if appropriate.

ACE XML Project files include an element corresponding to each of the four core ACE XML formats. These can be used to specify zero, one or many files of each type. The ACE XML parsing software automatically performs a data structure merge if multiple files of the same type are specified. Class Ontology files are the exception to this, since only one such file can be specified at a time because merging different ontologies can lead to significant inconsistencies if not supervised carefully. It is also possible to specify a path to a trained classification model and to a Weka ARFF file, as well as to multiple external resources via *uri* tags if desired. The Project file XML DTD is shown in Figure 12.

```
<!ELEMENT ace_xml_project_file_2_0 (comments?,
                                    feature_value_id,
                                    instance_label_id,
                                    class_ontology_id,
                                    feature_description_id,
                                    weka_arff_id?,
                                    trained_model_id?,
                                    uri?)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT feature_value_id (path*)>
<!ELEMENT instance_label_id (path*)>
<!ELEMENT class_ontology_id (#PCDATA)>
<!ELEMENT feature_description_id (path*)>
<!ELEMENT weka_arff_id (#PCDATA)>
<!ELEMENT trained_model_id (#PCDATA)>
<!ELEMENT uri (path*)>
<!ATTLIST uri predicate CDATA #IMPLIED>
<!ELEMENT path (#PCDATA)>
```

Figure 12: The XML DTD for the ACE XML 2.0 Project file format.

Although ACE XML Project files do make the combined use of multiple ACE XML files together more convenient, they are an imperfect solution on their own. Users must still maintain the individual files, and must be careful not to delete, rename or move them without making appropriate changes in the Project file.

The ACE XML ZIP file format is a solution to these problems. This solution involves packaging related sets of ACE XML (or other) files together into a single ZIP file. This has advantages with respect to reduced maintenance and increased convenience, while retaining the advantages of having separate files. Each component file stored in an ACE XML ZIP file remains self-contained and can be easily extracted from the ZIP file and used on its own or with other projects if desired.

Another significant advantage is that ZIP files are compressed formats, which means that they can dramatically reduce space and bandwidth requirements. This is significant, as ACE XML files can be quite large, particularly in cases where many windowed features are extracted from large collections of data.

jMIR's ACE libraries include a number of open-source utilities designed to facilitate the use of ACE ZIP files. Each ACE XML ZIP file is associated with an ACE XML Project file that serves as a key for accessing the files stored in the ZIP file, as well as their context. This Project file is either specified by the user upon creation of the ZIP file, in which case the associated ACE XML files are automatically added to the ZIP file when it is created, or auto-generated by the ACE XML utilities if ACE XML files are instead specified individually.