The M2K project

J. Stephen Downie

M2K is being developed to provide the Music Information Retrieval (MIR) community with a mechanism to access a secure store of copyright-sensitive music materials in symbolic, audio and graphic formats. *M2K* is a set of open-source, music-specific, *D2K* modules jointly developed by members of the *IMIRSEL* project and the wider MIR community. *M2K* modules include such classic signal processing functions as Fast Fourier Transforms, Spectral Flux, etc. In combination with *D2K*'s built-in classification functions (e.g., Bayesian Networks, Decision Trees, etc.), the *M2K* modules allow MIR researchers to quickly construct and evaluate prototype MIR systems that perform such sophisticated tasks as genre recoginition, artist identification, audio transcription, score analysis, and similarity clustering.

John Unsworth and **J. Stephen Downie** will lead a wrap-up and future work open-forum discussion: For ambitious, multi-institutional projects like those presented in this panel many issues arise that can affect the sustainability and impact of the projects. In particular, the issues surrounding the HC community's development, validation, distribution, and re-use of *D2K/T2K/M2K* modules will be addressed.

---

1. See <http://alg.ncsa.uiuc.edu/do/tools/d2k>.
2. See <http://www.news.uiuc.edu/news/04/1025mellon.html>.
3. <http://www.tapor.ca/>
4. See <http://music-ir.org/evaluation>.

A Declarative Framework for Modeling Pronunciation and Rhyme

David Dubin *(ddubin@uiuc.edu)*
University of Illinois
David J. Birnbaum *(djbpitt+@pitt.edu)*
University of Pittsburgh

Encoding standards such as TEI give scholars a great deal of flexibility in annotating texts to meet the particular needs of a study or project. Researchers necessarily make choices about which features of a text to highlight, what kinds of additional information to add, and what facts are left to be inferred from other sources of evidence apart from markup.

Among the factors to be considered in designing or adopting text encoding procedures are the prospects for:

- data reuse and generalization beyond the scope of the one's current project,
- investigating new questions about the texts that hadn't been anticipated, and
- planning for the integration of texts using other markup schemes.

This paper discusses considerations motivating the ongoing design of a software framework for analysis of rhyming schemes in 19th century Russian poetry. At its simplest, the application receives as input poems marked up like the example in Figure 1 and produces output like the following:

```
?- run('poem.xml').
Line 001 rhymes with Line 003
Line 002 rhymes with Line 004
Line 003 rhymes with Line 001
Line 004 rhymes with Line 002
```

```
<POEM OPID="S1.100" LINESPAN="1-4"
COLPAGE="1.261" YEAR="1817"
MASCRHYME="2" FEMRHYME="2"
OTHERRHYME="0" MASCUNRHYME="0"
FEMUNRHYME="0" OTHERUNRHYME="0"
NOENDWORD="0" COL13="2.75">
<TITLE>Надпись на стене больницы</TITLE>
<LINE LINENO="001">Вот здесь лежит
больной студ<STRESS>е</STRESS>нт;</LINE>
```

```
<LINE LINENO="002">Его судьба
неумол<STRESS>и</STRESS>ма.</LINE>
<LINE LINENO="003">Несите прочь
медикам<STRESS>е</STRESS>нт:</LINE>
<LINE LINENO="004">Болезнь любви
неизлеч<STRESS>и</STRESS>ма!</LINE>
</POEM>
```

Figure 1

Programs producing output such as the example above could be written in any of a variety of different programming languages. They might employ different strategies for integrating linguistic and orthographic processing rules with evidence encoded more directly using markup. For example, we discuss an earlier approach to the current project in Adams & Birnbaum. Our current implementation, however, is written in Prolog as an application of the BECHAMEL system for markup semantics analysis (Dubin et al.). The motivation for this choice was our wish to plan from the beginning for extensions to other encoding schemes and generalization to other kinds of analysis.

Prolog is a declarative language of rules and assertions, and BECHAMEL is a collection of predicates supporting the declaration of object classes, properties, relations among objects, and the execution of inference rules based on information extracted from XML documents. An example of a Prolog clause is shown below: it is part of our application's logic for determining that two sequences of phonemes at the ends of a pair of orthographic lines all agree with each other (i.e., that the sounds at the end of the lines rhyme with each other).

```
all_agree(P1,P2) :- agree(P1,P2),
/* P1 is written using C1 */
relation_applies(written,[P1,C1]),
relation_applies(written,[P2,C2]),
/* C3 follows C1 */
relation_applies(follows,[C1,C3]),
relation_applies(follows,[C2,C4]),
relation_applies(written,[P3,C3]),
relation_applies(written,[P4,C4]),
all_agree(P3,P4).
```

In the clause, P1, P2, P3, and P4 are variables representing phoneme objects, and C1, C2,C3, and C4 are variables representing character objects. The predicate all_agree(P1,P2) will be satisfied if each of the predicates following the implication sign can be satisfied. The logic of the clause can be read as follows:

> Phonemes P1 and P2 'all agree' if phonemes P1-P4 are written using characters C1-C4, respectively, if C3 follows C1 in the orthographic line, if C4 follows C2 in the line, if P1 and P2 'agree' and if P3 and P4 'all agree.'

This clause is one of several in a recursive rule that attempts to match up corresponding phonemes in rhyming lines, starting from the line's stressed vowel.

A major advantage of Prolog's declarative approach is the flexibility to define logic for separate cases in separate clauses for the same rule. For example, in the clause shown above, it is presupposed that in both lines there will be a simple one-to-one mapping from characters in sequence to phonemes. But accommodating more complex cases need not complicate the expression of the simple case: if the simpler clause cannot be satisfied then Prolog's inference engine will search for a different clause of the same rule that can be satisfied.

Reasoning about poems like the one in the example above requires that we model their contents at both a phonemic and orthographic level. The rules for Russian pronunciation include not only the way that particular vowels and consonants sound, but also how those phonemes are cued by the way the text is written (as with, for example, the palatalizing effect of the soft sign and the soft vowel letters). The BECHAMEL system's definition predicates for object classes, properties, and relations gives us the ability to model each of these levels with declarations such as the following:

```
/* There are things called phonemes */
declare_class(phoneme).
/* There are things called vowels */
declare_class(vowel).
declare_class(consonant).
/* vowels are a kind of phoneme */
declare_subclass(vowel,phoneme).
declare_subclass(consonant,phoneme).
/* vowels may be stressed or not */
declare_property(vowel,stress,atom).
declare_property(consonant,palatalized,atom),
declare_relation(written, [phoneme,
letter]),
declare_class(character),
declare_property(character,id,atom),
declare_property(character,palatalizing,atom),
```

In our approach, the phonemic properties of vowels and consonants are distinct from the orthographic properties of characters, written words, and lines. But it can occasionally be convenient for users to ignore these distinctions, particularly in comparing different proposed models for the same data. We therefore aim to let the models govern as much processing of our raw data as possible. For example, both phonemic and orthographic properties of letters are recorded in a data file using the same predicate as shown below:

```
alph_table(1083,id,u043B).
alph_table(1083,name,el).
alph_table(1083,charclass,consonant).
```

```
alph_table(1083,case,lower).
alph_table(1083,voicing,voiced).
alph_table(1083,place,alveolar).
alph_table(1083,manner,liquid).
```

In this example, `id`, `name`, `case`, and `charclass` are all properties of characters, while `voicing`, `place`, and `manner` are phonemic properties. As individual characters and phonemes are instantiated, they acquire only those properties that are appropriate for their class. This is accomplished through general-purpose rules that match on the basis of our property declarations. So if we were to decide (for example) that `name` should be a property of the phoneme rather than the character, we need only change the declaration, and the property value recorded in the data file would be assigned to phoneme objects rather than character objects.

BECHAMEL supports `superclass` and `subclass` relations, which allows us to declare that vowels and consonants are subclasses of `phoneme`, and that letters, marks, and spaces are subclasses of `character`. Since the conventional superclass/subclass relation can prove awkward in some situations, BECHAMEL includes class declaration expressions similar to those found in ontology languages such as OWL (W3C).

For example, place and manner of articulation in consonants may be used to describe classes of phonemes, not merely features of them (the class of stops, the class of velar consonants, etc.). It would be awkward to declare each consonant as a subclass of both its place and manner of articulation. Instead we use a BECHAMEL predicate that permits us to declare membership in a class based on the value assigned to a particular property:

```
declare_propclass(alveolar,place,alveolar).
declare_propclass(velar,place,velar).
declare_propclass(glottal,place,glottal).
declare_propclass(glide,manner,glide).
declare_propclass(liquid,manner,liquid).
declare_propclass(nasal,manner,nasal).
```

An alveolar, therefore, is anything that takes the value `alveolar` on its `place` property, a nasal anything that takes `nasal` for its `manner` property, and so on. These class identities are in addition to the one that instantiated the object. A related feature of BECHAMEL is the ability to define class membership based on a Boolean expression. The following example declares that an obstruent is either a stop, a fricative, or an affricate:

```
declare_boolean(obstruent,or(stop,
or(fricative,affricate))).
```

All of these features are employed with the aim of making our understandings, models, and simplifications regarding the rules of Russian pronunciation as clear and as explicit as possible. We express them in the form of declarative rules so as not to entangle implementation details of our code with aspects of our model that should be open to criticism, revision, and extension. For example, our rule governing the devoicing of word-final obstruents states that if an obstruent O is written with word-final letter L, then O should take a value of `voiceless` on its voicing property (unless it already has that property value):

```
mrule2 :- isa(O, obstruent),
relation_applies(written,[O,L]),
property_applies(L,wordfinal,true),
not(property_applies(O,voicing,voiceless)),
apply_property(O,voicing,voiceless),!.
```

Taking this approach, even a limited application, such as determining which lines of a poem rhyme, requires a large number of these declarations and rules; there is a very real sense in which we are doing it 'the hard way'. But as a result our application-specific code is only about seven percent of the size of the supporting declarations and rules. In addition to strengthening our confidence in our framework's potential for generalization, the demands of our approach have been a helpful modeling exercise in their own right.

Bibliography

Adams, L.D., and D. Birnbaum. "Perspectives on computer programming for the humanities." *Text Technology* 7.1 (1997): 1-17.

Dubin, D., C.M. Sperberg-McQueen, A. Renear, and C. Huitfeldt. "A logic programming environment for document semantics and inference." *Literary and Linguistic Computing* 18.2 (2003): 225-233.

W3C. *OWL Web Ontology Language Overview (W3C Recommendation).* 10 February 2004. Accessed 2005-04-04. <http://www.w3.org/TR/2004/REC-owl-features-20040210/>