BioC: A Minimalist Approach to Interoperability for Biomedical Text Processing

Donald C. Comeau[1], Rezarta Islamaj Doğan[1], Paolo Ciccarese[2,3], Kevin Bretonnel Cohen[4], Martin Krallinger[5], Florian Leitner[5], Zhiyong Lu[1], Yifan Peng[6], Fabio Rinaldi[7], Manabu Torii[6], Alfonso Valencia[5], Karin Verspoor[8], Thomas C. Wiegers[9], Cathy H. Wu[6], and W. John Wilbur[1]

[1]National Center for Biotechnology Information, [2]Massachusetts General Hospital , [3]Harvard Medical School, [4]University of Colorado School of Medicine, [5]Spanish National Cancer Research Centre, [6]University of Delaware Center for Bioinformatics & Computational Biology, [7]University of Zurich, [8]National ICT Australia, [9]Department of Biology at North Carolina State University

Supplementary material

Introduction

This is an accompanying document to the main manuscript of *BioC: A Minimalist Approach to Interoperability for Biomedical Text Processing.* BioC is an XML format and accompanying software to ease sharing of text corpora, data annotations, and NLP processing tools. Reading this document should follow the perusal of the main manuscript as the sections and details described here are closely related to the details described there. This document includes:

- Implementation and data classes
- Sample program
- Discussion
- Additional key file examples
- An additional relation example

Implementations

For this proposal to be practical there should be implementations in multiple languages. Implementations are now available in C++ and Java. These are two of the most commonly used languages for natural language processing.

Our goal with these implementations is that programmers can think of the data as existing solely in memory. Input connector classes make the data appear in internal data structures. Output connector classes take data out of internal data structures. While the data will be read from and written to XML files or network streams for portability to other languages and environments, ideally a programmer can

ignore XML while writing their program to process or prepare the data. This implementation aims to provide such functionality.

Simple classes store the collection and document information. Figure 1 and Figure 2 give skeletons of the BioC C++ and Java data classes. (Differences from the real code include: multiple public Java classes appear to be in the same file, accessors and comments are missing, etc.) These classes easily provide information for further processing. These classes are also easy to populate for exporting information. Because an XML library provides the processing, the programmer using BioC can ignore the quoting and other cautions that usually have to be considered when producing XML.

An important point regarding the data classes is that names of the data elements in the classes are the same as, or very similar to, the elements in the DTD that correspond to them. We believe this is important in reducing the cognitive burden in using this approach and should be followed as far as possible for other languages. (The Java classes begin with BioC because Collection and Document are common Java class names.)

In addition to a language, an XML parser has to be selected for an implementation. Several options are available. We chose libxml (http://www.xmlsoft.org/) for our C++ implementation because it is already available on many Unix machines and binaries are available for Windows. We use the C interface instead of the C++ interface because of a dependency on Gnome in the latter.

Libxml provides validation via an external program xmllint. This allows one to determine whether an XML file is valid for a DTD while still allowing the use of invalid files if they can be handled by a program. Most libraries that provide runtime validation have an option to determine whether or not validation should be performed.

We provide two complementary approaches to reading and writing data. For simplicity, we provide a set of methods that assumes an entire XML file can fit into memory at once. There the information is readily available for any analysis or processing needs. If information needs to be saved as XML, it should be organized using the same classes. Then it is converted to XML by the system and written out. On the other hand, for large collections it is unreasonable to hold the entire collection in memory. Most XML parsers have a serial interface that only requires a modest amount of information from the XML file to be in memory at one time. We also provide a set of methods that allow reading or writing a collection XML file one document at a time. This feature is important for large corpora.

For many XML parsers, the serial parser is implemented with either call backs or with a handler object. While this option is also offered by libxml, we use the interface that allows more natural IO under the direct control of the main program. In Java this is referred to as a Streaming API for XML (StAX) interface. At this time, we do not provide an interface using callbacks or handlers.

```
class Node {
  string refid;                 // id of Relation or Annotation
  string role;
};

class Relation {
  string id;
  map<string,string> infons;
  vector<Node> nodes;
};

class Location {
  int offset;
  int length;
};

class Annotation {
  string id;
  map<string,string> infons;
  vector<Location> locations;
  string text;
};

class Sentence {
  map<string,string> infons;
  int offset;
  string text;
  vector<Annotation> annotations;
  vector<Relation> relations;
};

class Passage {
  map<string,string> infons;
  int offset;
  string text;
  vector<Sentence> sentences;
  vector<Annotation> annotations;
  vector<Relation> relations;

};

class Document {
  map<string,string> infons;
  string id;
  vector<Passage> passages;
};

class Collection {
  string source;
  string date;
  string key;
  map<string,string> infons;
  vector<Document> documents;
};
```

Figure 1 Data members in BioC C++ classes.

```
public class BioCNode {
   protected String refid;    // id of Relation or Annotation
   protected String role;
}

public class BioCRelation {
   protected String id;
   protected Map<String,String> infons;
   protected List<BioCNode> nodes;
}

public class BioCLocation {
   protected int offset;
   protected int length;
}

public class BioCAnnotation {
   protected String id;
   protected Map<String,String> infons;
   protected List<BioCLocation> locations;
   protected String text;
}

public class BioCSentence {
   protected Map<String,String> infons;
   protected int offset;
   protected String text;
   protected List<BioCAnnotation> annotations;
   protected List<BioCRelation> relations;
}

public class BioCPassage {
   protected Map<String,String> infons;
   protected int offset;
   protected String text;
   protected List<BioCSentence> sentences;
   protected List<BioCAnnotation> annotations;
   protected List<BioCRelation> relations;
}

public class BioCDocument {
   protected String id;
   protected Map<String,String> infons;
   protected List<BioCPassage> passages;
}

public class BioCCollection {
   protected String source;
   protected String date;
   protected String key;
   protected Map<String,String> infons;
   protected List<BioCDocument> documents;
}
```

Figure 2 Data members in BioC Java classes.

```
#include <iostream>
#include "BioC.hpp"               // BioC data
#include "BioC_libxml.hpp"        // BioC libxml connector
#include "BioC_util.hpp"          // convenience base class for Sentence_Segmenter

using namespace std;
int main(int argc, char **argv) {

  if (argc != 3) {
    cerr << "Usage: " << argv[0] << " docname outname\n";
    return -1;
  }

  char * docname = argv[1];
  char * outname = argv[2];

  Collection collection;          // input data
  Connector_libxml xml;           // input connector
  xml.start_read(docname, collection);

  Sentence_Segmenter segmenter;   // process data
  Collection sentenceCollection;  // output data
  segmenter.convert( collection, sentenceCollection );
  sentenceCollection.key = "sentence.key";

  Connector_libxml xml_writer;    // output connector
  xml_writer.start_write( outname, sentenceCollection );

  Document document;
  while ( xml.read_next(document) ) {
    Document sentenceDocument;
    segmenter.convert( document, sentenceDocument ); // program processing
    xml_writer.write_next( sentenceDocument );
  }

  xml_writer.end_write();

  return 0;
}
```

Figure 3 Sample C++ main program.

Sample Program

A sample program appears in Figure 3. Note that it includes a header for the BioC data classes and a header for an XML connector class. Data objects are declared both for input and output. In addition to the Collection objects for overall information, Document objects were used because the program uses document-at-a-time IO. Connectors for both input and output were declared so information for both the input and output XML files could be maintained. The data is processed by an overloaded method of a class. It could be handled by arbitrary code. In fact, the data could be read at the beginning of a pipeline, passed through many stages in an internal format and the desired information written by a final program.

There are two Java BioC implementations. One uses the standard XML processor and the other uses he WoodStox XML processor (http://woodstox.codehaus.org/). Both use a StAX interface so flow control is conventional. Both implement whole file and document at a time IO.

The clear division between the classes that enclose our XML text data, the code that interacts with the XML parser, and our application code is an important feature of our sample program. Changing the data

processing code is sufficient to create a new application. Using a different XML parser would only require changing the XML connector class. This ensures the code would easily fit into any environment. This arrangement was depicted earlier in Figure 1 in the manuscript. The connectors, the data classes, and the data processing are important parts of the program or pipeline. Yet they can be changed or modified without disrupting the overall flow. This allows great flexibility so a program can do what is needed. Yet the program will be able to work with a standard format and share the results in a standard manner.

The BioC XML files only include data, not processing instructions or directions. As in our example, we expect that to be provided via another channel to the data processing module of a program or system. Adequately describing and encapsulating the data is challenge enough. We do not expect to be able to describe all the creative uses for that data.

Discussion

The BioC XML format is not intended as a replacement for the internal data structures of any research group. Internal methods and pipelines allow internal optimization of the processing. BioC ensures data exchange and interoperability, by allowing data to flow easily between different systems, platforms and software arrangements.

We recommend separate files for a collection and annotations based on that data. If annotations of different entity types were obtained from different sources, this would be expected. Keeping the original text document separate from the annotation files would make it easier to verify that annotations correspond to the same base text. However, the BioC DTD is flexible and allows text and annotations to be combined in the same file if necessary.

When using separate text and annotation files, information linking the annotations with the original text document file should be included in the annotation key file. There are varying ways of connecting documents and annotations. For example, files could be allocated in the same directory or webpage location, etc.

Since XML is an easily comprehended format, it is often expected that XML files are formatted to increase human readability. When text contains nested elements, this formatting can change the meaning of the file because of the added whitespace. While the current BioC data model does not contain such nested elements, current BioC XML tools do not format their output to avoid potential problems. XML formatters and editors can produce a more readable XML file, as displayed in the examples in the main manuscript.

The character set is Unicode. Unicode allows the explicit and unambiguous inclusion of national character sets, mathematical symbols, and many other glyphs that appear in published material. UTF-8 is the most convenient encoding. It is the same as ASCII for 7 bit characters. This is one of the encodings required to be implemented by XML tools and is portable between big-endian and little-endian machines. Code points beyond 127 may be expressed directly in UTF-8 or indirectly using numeric entities.

While Unicode is growing in use and application, many natural language tasks are still performed in ASCII. For such tasks there is no value spending the extra resources or effort on the additional complexity of Unicode. Converting English text from Unicode to ASCII should be a standard operation with well-defined translation tables. The conversion program we have implemented uses an internal library method and internal data. There could be standard, public tables and a simple program to apply these translations even though such a program would be less efficient. The collection key file should state the text encoding.

While our attention is on Unicode, there exist myriad other encodings and useful text corpora expressed using these other encodings. These corpora might become more popular if converted into Unicode. In the meantime, they can be expressed directly in the BioC framework, but, the accompanying key file should explicitly state the used encoding.

Offset specification is important for linking textual annotations to their original locations in the text. The offset of the annotated string in the original text string is dependent on the encoding of that text string. For ASCII and/or UTF-8 encodings byte offsets provide a logical approach. More generally, with a known encoding, the number of code points preceding a given point in the text provides an unambiguous reference to that point in the text and could be used to indicate the offset. These specifications must be clearly stated in the accompanying key file.

Key file examples

```
This key file describes the contents of the BioC XML file exampleCollection.xml.

collection:            10 arbitrary PubMed documents in Unicode
   source:             PubMed
   date:               yyyymmdd. Date documents downloaded from PubMed
   key:                this file

   document:           Title and abstract (if available) from a PubMed reference
      id:              PubMed id

      passage:         Either title or abstract
         infon type:   "title" or "abstract"
         offset:       PubMed is extracted from an XML file, so literal offsets would not be
                       useful. Title has an offset of zero, while the abstract is assumed to
                       begin after the title and one space. These offsets at least sequence
                       the abstract after the title.
         text:         The original Unicode text as obtained from the PubMed XML
```

Figure 4 The exampleCollection.key file describing basic elements such as collection, document and passage.

```
This key file describes the contents of the BioC XML file asciiCollection.xml.

collection:            10 arbitrary PubMed documents with all text ASCII
   source:             PubMed
   date:               yyyymmdd. Date documents downloaded from PubMed
   key:                this file

   document:           Title and abstract (if available) from a PubMed reference
      id:              PubMed id

      passage:         Either title or abstract
         infon type:   "title" or "abstract"
         offset:       The original Unicode byte offsets were retained.
         text:         The original Unicode text converted to ASCII using the IRET Unicode to
                       ASCII conversion.
```

Figure 5 The asciiCollection.key file describing the same basic elements of collection, document and passage, after Unicode text conversion to ASCII

```
This key file describes the contents of the BioC XML file asciiSentence.xml.

collection:            10 arbitrary PubMed documents with all text ASCII split into sentences
                       by the MedPost sentence segmenter
   source:             asciiCollection.xml
   date:               yyyymmdd. Date documents downloaded from PubMed
   key:                this file

   document:           Title and abstract (if available) from a PubMed reference
      id:              PubMed id

      passage:         Either title or abstract
         infon type:   "title" or "abstract"
         offset:       The original Unicode byte offsets were retained.

         sentence:     One sentence of the passage as determined by the MedPost sentence
                       splitter
            offset:    A document offset to where the sentence begins.
            text:      The ASCII text of the sentence.
```

Figure 6 The asciiSentence.key file describing basic elements such as collection, document, passage and sentence.

```
This key file describes the contents of the BioC XML file abbrv.xml.

collection:                  10 arbitrary PubMed documents with all text ASCII split into
                             sentences by the MedPost sentence segmenter
   source:                   asciiCollection.xml
   date:                     yyyymmdd. Date documents downloaded from PubMed
   key:                      this file

   document:                 Title and abstract (if available) from a PubMed reference
      id:                    PubMed id

      passage:               Either title or abstract
         infon type:         "title" or "abstract"
         offset:             The original Unicode byte offsets were retained.

         annotation:         Abbreviations
            id:              sequential integers from 0 within each passage
            infon type:      "Long Form" or "Short Form"
            location offset: A document offset to where the short form or long form begins.
            location length: The length of the short form or long form.
            text:            Original text of the short form or long form.

         relation:           Long form / short form pair
            id:              Relation identification string, R[\d]+, where R stands for
                             Relation and the number counts from zero within a passage
            infon type:      "abbreviation"
            node role:       "Short Form" or "Long Form"
            node ref_id:     id of the Short Form or Long Form annotation as appropriate
```

Figure 7 The abbreviationExample.key file illustrating annotations and relations in a passage.

Relation examples

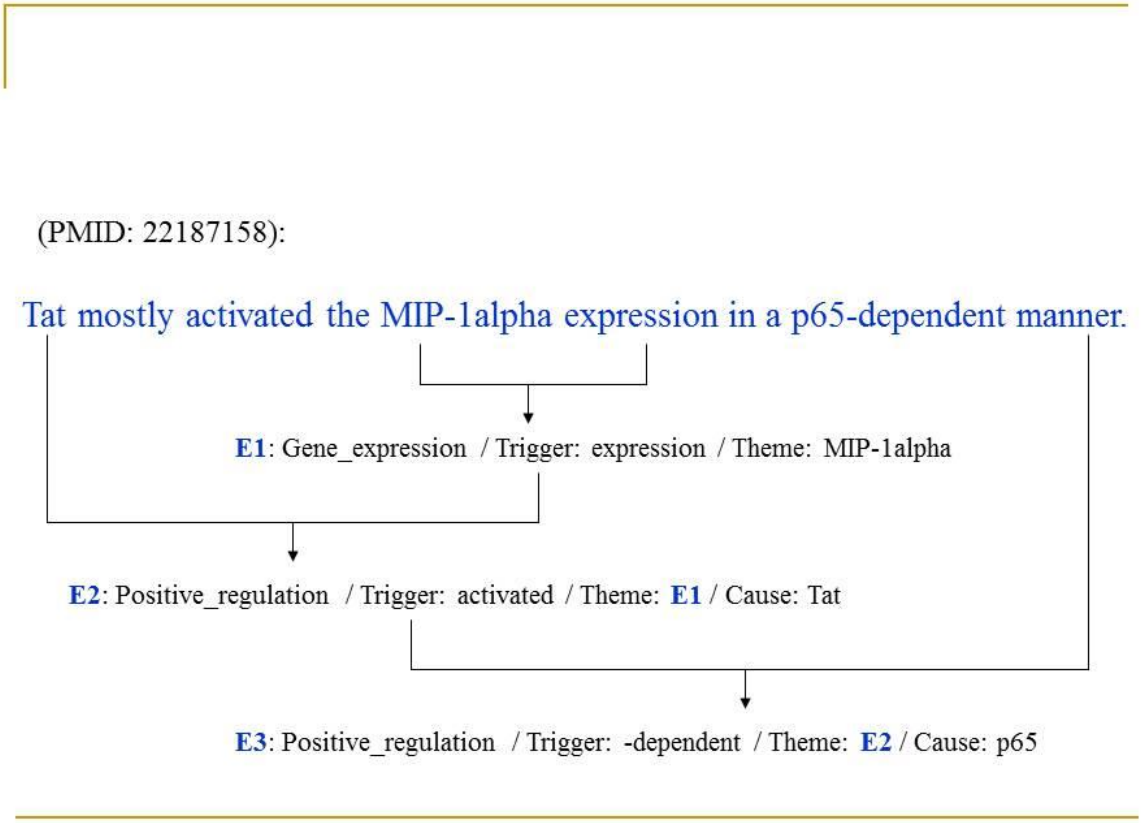

Figure 8 Sentence excerpt from a PubMed abstract illustrating nested protein-protein interaction events.

Figure 8 shows a short sentence from a PubMed article which includes three nested events. Notice the annotations of two different entity types: gene names and trigger words. These could potentially come from different annotation projects. The events are represented as relations in BioC and involve both entity types. The annotations require unique reference id's to permit correct referencing by nodes participating in a relation. Relations will usually contain multiple nodes, and any relation with an id may be a participant in other relations. Here we illustrate how the entities and events of Figure 8 are expressed as annotations and relations in BioC:

```
<annotation id =”G0”>
   <infon key=”type”>Gene_name</infon>
   <location offset=”0” length=”3” />
   <text>Tat</text>
</annotation>
<annotation id =”G1”>
   <infon key=”type”>Gene_name</infon>
   <location offset=”25” length=”10” />
   <text>MIP-1alpha</text>
```

```
</annotation>
<annotation id =”G2”>
   <infon key=”type”>Gene_name</infon>
   <location offset=”52” length=”3” />
   <text>p65</text>

</annotation>

<annotation id =”T0”>
   <infon key=”trigger”>Positive_regulation</infon>
   <location offset=”11” length=”9” />
   <text>activated</text>
</annotation>
<annotation id =”T1”>
   <infon key=”trigger”>Gene_expression</infon>
   <location offset=”36” length=”10” />
   <text>expresison</text>
</annotation>
<annotation id =”T2”>
   <infon key=”trigger”>Positive_regulation </infon>
   <location offset=”55” length=”10” />
   <text>-dependent</text>
</annotation>


<relation id="R0">
   <infon key =”event-type”>Gene_expression</infon>
   <node refid="G1" role="Theme"/>
   <node refid="T1" role="Trigger"/>
 </relation>
<relation id="R1">
   <infon key =”event-type”>Positive_regulation</infon>
   <node refid="R0" role="Theme"/>
   <node refid="T0" role="Trigger"/>
   <node refid="G0" role="Cause"/>
 </relation>
<relation id="R2">
   <infon key =”event-type”>Positive_regulation</infon>
   <node refid="R1" role="Theme"/>
   <node refid="T2" role="Trigger"/>
   <node refid="G2" role="Cause"/>
 </relation>
```