Extending BioC Implementation to More Languages

Wanli Liu, Donald C. Comeau, Rezarta Islamaj Doğan, and W. John Wilbur*

National Center for Biotechnology Information, National Library of Medicine, National Institutes of Health, 8600 Rockville Pike, Bethesda, MD 20894, MD, USA

*Corresponding author: Tel: 301 435 5926, E-mail: wilbur@ncbi.nlm.nih.gov

Abstract
As part of a community-wide effort for evaluating text mining and information extraction systems applied to the biomedical domain, BioC is focused on the goal of interoperability: currently a major barrier to wide-scale adoption of text mining tools. BioC is a simple XML format, specified by DTD, for exchanging data for biomedical natural language processing. With initial implementation in C++ and Java, BioC provides libraries of code for reading and writing BioC text documents and annotations. We extend BioC to scripting languages (Perl and Python) with SWIG, as well as to the Go language. BioC modules are functional in these languages, which can facilitate some BioCreative tasks. We also discuss the addition of new languages to support BioC in the future. BioC implementations are freely available at the BioC site: http://bioc.sourceforge.net.

Introduction
BioCreative Workshops provide the forum for text mining, computational linguistics and natural language processing researchers to build, adapt and/or integrate information extraction systems that address biologically meaningful tasks and that provide results of practical relevance. In order to ensure satisfactory community assessment and method comparison, and to promote scientific progress, it is necessary to establish common standards and shared criteria that enable comparison and integration of different approaches. BioC (1) has been gaining momentum as a solution to the interoperability challenge — an interchange format for biomedical natural language processing tools. Expressed in a simple XML format, specified by DTD, text documents and related data annotations can be shared easily between different text mining and information extraction systems applied to the biomedical domain. Moreover, all tools that communicate with this shared format can potentially be combined as parts of larger, more complicated systems.

Considering the variety of computational tools employed by the biomedical text mining community, successful interoperability requires uniform BioC support across various programming language environments. The first releases of BioC code were implemented in C++ and Java, two mainstream programming languages which provide a solid foundation for defining
BioC functionality. However, scripting languages such as Perl and Python have become popular in the communities of bioinformatics and natural language processing for their ease of use with reasonable performance. For example, BioPerl\(^1\) and Biopython\(^2\) are widely adopted for the tasks of parsing BLAST output and querying the GENBANK database (2,3). Go\(^3\), a newly-emerging language from Google, is also receiving attention with biogo\(^4\) targeted at computationally intensive Bioinformatics tasks. For biomedical natural language processing tasks, a variety of NLP packages (CPAN-NLP\(^5\), NLTK\(^6\)) have been developed with Perl and Python. Therefore, it is important to make BioC functionalities easily accessible for applications coded with these languages. With the support of BioC modules, applications are able to conveniently extract information from BioC XML files, process the information, and write the output in BioC XML format. This facilitates efficient and uniform data sharing. Directly implementing BioC in a large number of other languages would require a large amount of work and additional exhaustive testing. To efficiently deliver behavior identical to C++ in scripting languages, we employ the interface compiler SWIG\(^7\) to connect the C++ BioC implementation to the target languages: Perl and Python. By supporting full BioC functionality faithfully, the integration of the C++ BioC implementation and scripting languages enables fast and flexible prototyping while relying on the low-level C++ code, to duplicate the behavior of C++ applications. In this paper we introduce the effort of our team to enable BioC users to take advantage of BioC utilities in scripting language applications with some examples explained in detail, as well as our experiments with the Go language.

**BioC Interface with Scripting Languages**

To reach high interoperability and reusability in NLP and text processing tasks, BioC is designed as a simple workflow based on XML formats (1). As in the C++ and Java environments, the BioC workflow in a scripting language context uses two connector modules for XML input/output, as shown in Figure 1. In Perl or Python, an Input Connector object is created to gain access to XML input from a file or network stream. This input is converted to data encapsulated in BioC data classes. The BioC data classes then provide various methods to retrieve data for the data processing stage, which is implemented in a scripting language. Modify methods of BioC data classes can be used to update BioC data classes, or new data can be created. Finally, the output connector writes the new or updated data in XML format.

As shown in Figure 1, both BioC data classes and Input/Output connectors are coded in C++, while the data processing stage is coded in a scripting language. Data travels between C++ and

\(^1\) http://www.bioperl.org
\(^2\) http://biopython.org/
\(^3\) http://golang.org/
\(^4\) https://code.google.com/p/biogo/
\(^5\) http://cpan.org
\(^6\) http://nltk.org
\(^7\) http://www.swig.org
scripting language contexts through BioC class methods. The separation of the data processing stage and the Input/Output connectors enables the data processing stage to focus on processing biomedical data, regardless of the XML format of the Input/Output files. As in C++ applications, input stage, data processing stage, and output stage can be decoupled as needed.

![BioC workflow for scripting language application](image)

**Figure 1.** BioC workflow for scripting language application

### A BioC C++ SWIG Interface
To obtain seamless data flow in BioC, it is critical to initialize Input/Output connectors, and communicate via methods of BioC classes in the target languages. To accomplish this, the BioC C++ header files (*BioC.hpp, BioC_libxml.hpp, BioC_util.hpp*) are processed by SWIG, which extracts the declarations in these header files to create wrapper codes in both the target scripting language and C++. The wrapper code in the target language defines proxy classes for the underlying C++ classes, as well as a variety of customization to suit the specific target language features. The original BioC source code is compiled by a C++ compiler to generate object file, in the same way as building a pure C++ application. These BioC objects are linked with an object compiled from C++ wrapper code to provide BioC proxy classes for the target language. The proxy classes can be accessed in the target language to provide methods to manipulate the data contained in the C++ BioC classes, as shown in Figure 2.
Using BioC in Scripting Languages

To illustrate the key points of using a BioC module in target languages, we use Perl code as an example (Figure 3). Based on BioC C++ version 1.0\textsuperscript{8}, SWIG version 2.0.4\textsuperscript{9} produces BioC_Perl.so and BioC_Perl.pm. BioC_Perl.pm contains the Perl proxy classes for the BioC C++ classes, and BioC_Perl.so provides the executable implementation of BioC classes.

![Diagram showing access to C++ BioC Class through target language proxy class wrapper interface](image)

**Figure 2.** Access C++ BioC Class through target language proxy class wrapper interface

After importing the Perl BioC module, we are ready to initialize BioC wrapper objects. The Connector_libxml class object ($xml) enables opening an XML file and reading the contained data into a Collection class object ($collection). Then we can iterate through the BioC XML file by visiting each document ($dcm) within the collection, and each passage ($psg) within each document, and each annotation ($ann) and relation ($rel) within each passage. While iterating through the XML file, the data members defined in BioC classes (e.g., {id} of Document class) can also be accessed directly or through methods via Perl wrapper objects. In addition to reading the whole XML file into ($collection) prior to processing, BioC also provides a read_next(document) method for one-document-at-a-time access.

In addition to the data read into the BioC classes, new data can be added to the BioC classes. One such example is the {infons} data structure in a number of BioC classes, which is a C++ std::map container template mapping a key string to a value string. The mapped value string can be retrieved via the get() method of the {infons} data structure with ELEMENT KEY string (and updated via the set() method).

After data is extracted from XML format and then processed by Perl code, the original collection class object can be modified in a fashion similar to reading BioC data classes. The updated

\textsuperscript{8} http://sourceforge.net/projects/bioc/files/BioC_C\%2B\%2B_1.0.tar.gz/download
\textsuperscript{9} http://sourceforge.net/projects/swig/files/swig/swig-2.0.4/
collection class object can be saved in XML format, ready to be accessed by another BioC application. The Perl BioC module has been successfully used by the NatLab abbreviation system (4) to make it BioC compatible (5).

```perl
# import BioC module from PATH TO PERL BIOC MODULE, where
# BioC_Perl.pm and BioC_Perl.so are located
BEGIN {push (@INC, PATH TO PERL BIOC MODULE);}  
use BioC_Perl;

my $collection = new BioC_Perl::Collection();
my $xml = new BioC_Perl::Connector_libxml();
$xml->read (XML INPUT, $collection);

# iterate through all documents
for ( my $i = 0; $i < $collection->{documents}->size(); $i++) {
  my $dcm = $collection->{documents}->get($i);
  print "$dcm->{id}\n";
  print "$dcm->{infons}->get(ELEMENT KEY)\n";
  # iterate through all passages
  for ( my $j = 0; $j < $dcm->{passages}->size(); $j++) {
    my $psg = $dcm->{passages}->get($j);
    # iterate through all annotations
    for ( my $k = 0; $k < $psg->{annotations}->size(); $k++) {
      my $ann = $psg->{annotations}->get($k);
    }
    # iterate through all relations
    for ( my $k = 0; $k < $psg->{relations}->size(); $k++) {
      my $rel = $psg->{relations}->get($k);
    }
  }
}
... # process data and modify $collection if necessary
$xml->write (XML OUTPUT, $collection);
```

Figure 3. Perl code accessing BioC module (tested with Perl 5.8.8)

Figure 4 shows Python code executing the same task as Perl code. Compared to Perl, BioC data classes interface with native Python structures more naturally, which improves convenience and ease of development. Unlike in Perl, the C++ std::map container is treated as a Python dict object, and a mapped string can be accessed as dict[key]. The example code demonstrates several ways to loop over BioC data. However, the stronger support in Python is still not complete. SWIG has not implemented all Python methods for the underlying C++ data structures. For example, the get() method returning the default value for a missing key in a Python dict object is not available in the SWIG implementation (although one can work around this particular issue by using the key in dict expression, which is supported). Another pitfall in applying our Perl and Python BioC modules has to do with the way the underlying C++ data is accessed. When updating the BioC data, references should be used in order to commit the new value (as shown in the code examples). SWIG sometimes implicitly
make a copy of the C++ data, and as a result, the updates or changes to the Python or Perl copy of the data are not reflected in the C++ data and may be lost. Care must be taken when updating or changing data to ensure the original C++ data is changed. A better practice may be to create new data objects with updated or new data when memory capacity permits.

```python
# import BioC module from PATH TO PYTHON BIOC MODULE, where # BioC_Python.py and _BioC_Python.so are located sys.path.append(PATH TO PYTHON BIOC MODULE) import BioC_Python

collection = BioC_Python.Collection()
xml = BioC_Python.Connector_libxml()
xml.read (XML INPUT, collection)

# iterate through all documents with iterator dcm (read only)
for dcm in collection.documents :
    print dcm.id
    print dcm.infons[ELEMENT KEY]
    # iterate through all passages
    for index, psg in dcm.passages:
        # iterate through all annotations with reference
        # (read/write)
        for k in range(0, psg.annotations.size()):
            ann = psg.annotations[k]

        # iterate through all relations with iterator (read only)
        for rel in psg.relations:

            ... # process data and modify $collection if necessary
xml.write (XML OUTPUT, collection)
```

Figure 4. Python code accessing BioC module (tested with Python 2.5.1)

**A BioC Go Implementation**

Go is a new language from Google developed by Ken Thompson and Rob Pike, who are known for UNIX, C, and Plan 9. Its features include the convenience of type inference and the goroutines for concurrency. As their web page\(^\text{10}\) says, “It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language.”

In Go, XML data can be marshaled and unmarshaled quite simply by using struct tags. One limitation is that there is no direct way to exchange information between XML and a map, as used to implement infons in other languages. So Go BioC objects hold both an Infons map and an InfonStructs slice. The XML data is marshaled and unmarshaled out of and in to the InfonStructs slice. The data is moved in to or out of the Infons map after reading or before writing. This is a small inefficiency because the amount of infon data is small. Reading or

\(^{10}\) [http://golang.org/doc/](http://golang.org/doc/)
writing a BioC collection a document at a time does require lower level interaction with the XML parser for the collection itself. The individual documents can still be written and read with the direct Marshal and Unmarshal functions. So far we are pleased with the language and expect to continue our experiments.

Conclusions
We describe the extension of BioC beyond C++ and Java programming languages, enabling convenient handling of BioC XML documents in other languages. SWIG allows the original behavior of C++ BioC classes to be available in scripting language contexts by reusing the same C++ source codes. However, different languages have different features and capabilities, which are exposed to varying extents by SWIG\textsuperscript{11}. Python is demonstrated to offer a much richer interaction with the C++ data structures than is available in Perl. Future versions of SWIG may provide more support for additional language features. A native implementation provides more natural interaction and integration with the language. However, it requires more effort than using SWIG and requires testing to guarantee consistent behavior with other BioC implementations. We are interested in suggestions from BioC users regarding additional languages, such as Ruby, and new programming features. Perl, Python, and Go BioC implementations are freely available at the BioC website.

Acknowledgements
This research was supported by the Intramural Research Program of the National Institutes of Health, National Library of Medicine.

References


\textsuperscript{11} \url{http://www.swig.org/Doc2.0/}

37