Structured Data Representation Using Ruby Syntax

KAZUAKI MAEDA
Department of Business Administration and
Information Science, Chubu University
1200 Matsumoto, Kasugai, Aichi
JAPAN
kaz@acm.org

Abstract: This paper describes Ribbon (Ruby Instructions Becoming Basic Object Notation), a new representation written in a text-based data format using Ruby syntax. The design principle of Ribbon is good readability and simplicity of structured data representation. An important feature of Ribbon is an executable representation. Once Ribbon-related definitions are loaded into a Ruby interpreter, the representation can be executed corresponding to the definitions. Java programs are expected to read/write Java objects to persistent storage-media, or to traverse the structured data. A program generator was developed to create Ruby and Java programs from Ribbon definitions. In the author's experience, productivity was improved in the design and implementation of programs that manipulate structured data.

Keywords: Data Representation, Structured Data, Domain Specific Languages, Ruby, Java

1 Introduction

Structured data has been widely used in many software development projects. In the case of compiler development, compiler front-ends build abstract syntax trees (ASTs), which represent structured syntactic information of source code[1]. Initially, "Diana" was designed as an AST for Ada programs[2], making it suitable for an intermediate representation of source code in Ada compilers.

A variety of representations for structured data have been developed to date. As a practical application of Diana, we can use an interface description language (IDL)[1] compiler in a Scorpion toolkit[4]. IDL is used to define structured data; the IDL compiler reads the Diana specification in the IDL and generates useful functions to manipulate an AST in the C programming language, including a writer to translate the AST to an external representation and a reader to reconstruct the AST from the external representation. The Scorpion toolkit is very useful in developing language-oriented software tools; however, the representation of structured data for the toolkit is not designed for general purposes. It is only specific to the toolkit; moreover, it does not support object-oriented programming languages.

Object serialization is included in standard Java packages[5], and it supports the translation of objects (and objects reachable from them) into a byte stream, as well as supporting the reconstruction of objects from the byte stream. The object serialization in Java is useful for saving temporal objects to persistent storage media such as hard disks. The author has developed a multi-pass compiler composed of front- and back-end programs. The front-end program explicitly invokes a method for object serialization, and writes the resulting AST to a file using this serialization, after which the back-end program reads the file and reconstructs the AST. The value of each field in the objects is saved with its type information so that, if necessary, the objects can be reconstructed from the saved file.

The object serialization in Java is helpful in writing out the state of objects; but at least two problems are evident. First, the serialized objects in Java are written in a binary format, the detailed information of which is not open. Second, the serialized objects cannot be used in other programming languages due to the lack of libraries for such languages that can read this type of data.

An attractive alternative for the representation of structured data is XML. XML is designed as an open, tag-based, and text-based data format; its specifications are managed by an open standards organization. It can be used across different platforms, i.e., different computers, operating systems, and programming languages. Many XML standards and technologies exist, ensuring an advantage over other data formats. However, the large number of XML standards and technologies create a barrier to learning XML and developing XML applications.

[1]The IDL in this context is different from OMG IDL[3].

This paper describes Ribbon[2], a new representation of structured data in a text-based data format using Ruby syntax. The author believes that good readability and simplicity of structured data representation are important for all developers. Ruby is a simple but powerful programming language, and its syntax is suitable for representation; therefore, Ribbon was designed in Ruby syntax.

Ribbon and its related tools support all object-oriented programming languages. Although Java was used in our study, Ribbon does not impose any restrictions on the programming language. C# or Visual Basic could also have been used.

An important feature of Ribbon is that the representation is executable. An XML parser is needed for XML application development. However, in the case of Ribbon, the parser for the representation is embedded in the Ruby interpreter; we therefore do not need to parse the representation in Ribbon. If Ribbon-related definitions are loaded into the Ruby interpreter, the representation can be executed corresponding to the definitions. This is useful for Java programs in reconstructing Java objects or for traversing the structured data.

Section 2 describes technologies for structured data representation. Section 3 explains Ribbon and its related tools, and Section 4 summarizes this paper.

2 Related Technologies

2.1 XML and DOM

XML is a markup language that controls the formatting and presentation of documents. Instead of the document-centric view, the data-centric view advocates that XML is suitable for applications requiring data exchange. Data can be exchanged across languages, processes, computers, and vendors. If source code is analyzed and the result is written in XML to represent syntactic information, the development of software tools is possible using various programming languages and libraries to manipulate the XML representation.

JavaML[6] is a typical XML-based source code representation for ASTs, providing syntactic and semantic information after parsing Java source code. Once software tools are implemented using the JavaML representation, they can easily obtain information about Java source code without the need for analysis. XSDML[8] and srcML[9] are other representations in XML that support the representation of formatting information, including white spaces and comments, in addition to ASTs. Therefore, the original source code is restored from the XML representation using the formatting information in XSDML or srcML.

The XML representation is parsed to a hierarchical tree of elements and other XML entities for manipulation in the main memory. After construction of the tree, each node can be accessed using tree traversal APIs. The document object model (DOM) was defined for standard tree access. The DOM-based approach is useful for programmers in manipulating the XML representation, according to the hierarchical tree structure. Programs using DOM traverse the structure; however, conversion from the document-oriented DOM to application-specific and problem-oriented data is necessary. If there are many kinds of objects to be converted, the work is tedious and error prone.

An XML representation is just static data, while the representation in Ribbon is dynamic, executable code. Once a representation in Ribbon is successfully loaded into a Ruby interpreter, the parsing of the data is already done. After being loaded, the representation can be executed into the Ruby interpreter; therefore, conversion from document-oriented representation to problem-oriented representation is unnecessary. It is useful in reconstructing Java objects from the representation.

2.2 DSL and Ruby

Domain specific languages (DSLs) are programming languages tailored to specific application domains[10] and designed precisely to describe problems in specific application fields[11]. They are special-purpose, and not general-purpose, programming languages.

Fowler has explained the difference between external and internal DSLs[12]. An external DSL (e.g., an XML configuration file) is a special-purpose language with a syntax that is different from existing programming languages. In case of an external DSL, the DSL developer has to build a parser for the domain-specific description. On the other hand, an internal DSL uses the constructs of the existing programming language (or “host” language) to define the DSL. In case of an internal DSL, the designer extends the host language to the domain specific description, which can improve the development time for building a DSL processor to execute the description.

Ruby is one of many general-purpose, object-oriented programming languages, combining scripting syntax with Smalltalk-like object-oriented features. The official implementation is written in C, but there are a variety of implementations of Ruby interpreters including Rubinius[14], JRuby[15], and

[2]Ribbon stands for Ruby Instructions Becoming Basic Object Notation.

IronRuby[16].

Ruby plays an important role as a host language for an internal DSL. One of Ruby's appealing features is the fact that parentheses for arguments of methods are optional; therefore, the descriptions are easier to read and understand than ones in other programming languages. Another feature in Ruby is "block," a group of program statements; it can be an argument of a method in Ruby. The block is powerful in its representation of structured data. Ruby's features are advantageous to Ribbon in representing structured data. Ribbon uses these advantages of Ruby to represent structured data. The next section explains the details of Ribbon.

3 Representation and Manipulation of Structured Data

3.1 Ribbon as an Object Notation

The representation of structured data in Ribbon is composed of several elements. Each element has a name and a value. For example,

```
config "my behavior"
```

which represents the name of the element as *config* and the value as "my behavior." The element *config* is not just a data representation, but internally, it is also an executable method invocation without parentheses in Ruby. The method name is *config* and the argument of the method is "my behavior."

In Ruby, an element can have child elements using blocks. For example, Figure 1 shows that the *config* element has three child elements: *before*, *buy*, and *at*. The *before* element's value is "programming," the *buy* element's value is "coffee," and the *at* element's value is "coffee stand." Basically, one line can only have one element, according to Ruby syntax. If we write a semicolon at the end of an element, we can use multiple elements in one line. This action is not recommended, however, since the author believes that simplicity is very important in representing structured data in Ribbon.

```
config "my behavior" do
  before "programming"
  buy "coffee"
  at "coffee stand"
end
```

Figure 1: An element with three child elements

Ribbon supports four primitive data types: *int*, *real*, *bool*, and *string*. For example, Figure 2 shows

```
config "my behavior" do
  pay "today" do
    price 340
    tax  0.05
    togo true
    date "April 1, 2008"
  end
  before "programming"
  buy "coffee" do
    order "grande"
    order "hot"
    order "double-shot"
  end
  at "coffee stand"
end
```

Figure 2: Elements including primitive data types and a collection

that the *pay* element's value is "today," and it has four child elements: *price*, *tax*, *togo*, and *date*. The *price* element has an integer value 340, the *tax* element has a real value 0.05, the *togo* element has a boolean value true, and the *date* element has a string value "April 1, 2008."

An element in Ribbon cannot have more than one child element with the same name. Recall that a Java class has no more than one field with the same name. An element in Ribbon is a similar construct as a Java class. In comparison, an element to represent a collection can have more than one element with the same name. In Figure 2, the *buy* element has three child elements with the name *order*. The first *order* element's value is "grande," the second *order* element's value is "hot," and that of the last is "double-shot." This represents a sequence of elements.

If we need to represent an element linked to another element across the structure, a unique identifier is given to the element, and another element refers to the element using the identifier. Figure 3 shows that a uuid element with the identifier dbe0c2e6_5a63_4159_a76f_6e44 is given to the *store* element with "Seattle, Washington." The identifier in the figure is calculated using java.util.UUID, which is represented using a symbol in Ruby; the *at* element has a reference to the unique identifier. This means that Ribbon supports graph-structured data.

3.2 Definition of Structured Data

Structured data is defined using symbols in Ruby and the four keywords *has*, *is_a*, *is_seq_of* and *is_type*, as shown in Table 1.

Figure 4 shows the definitions of the representa-

```
location "coffee stand" do
  store "San Francisco, California"
  store "Seattle, Washington" do
    uuid :dbe0c2e6_5a63_4159_a76f_6e44
  end
end
config "my behavior" do
  before "programming"
  buy "coffee"
  at :dbe0c2e6_5a63_4159_a76f_6e44
end
```

Figure 3: Elements including a uuid and the reference

Table 1: Keywords to define structured data

keyword	meaning of the keyword
has	composition of elements
is_a	specialization of an element
is_seq_of	collection of elements
is_type	primitive data type (int, real, bool or string)

tion in Figure 2. The definitions are as follows:

- *config* element has four child elements: *pay*, *before*, *buy*, and *at*
- *buy* element is a collection of *order* elements
- *pay* element has four child elements: *price*, *tax*, *togo*, and *date*
- *price* element has an integer value
- *tax* element has a real value
- *togo* element has a boolean value
- *date* element has a string value

```
:config.has :pay, :before, :buy, :at
:buy.is_seq_of :order
:pay.has :price, :tax, :togo, :date
:price.is_type :int
:tax.is_type :real
:togo.is_type :bool
:date.is_type :string
```

Figure 4: An example of Ribbon definitions

3.3 Program Generation for Ruby and Java

The representation in Ribbon becomes executable if Ribbon-related programs are loaded into a Ruby interpreter. For example, Figure 5 shows a program to display only the value of *config* element in Figure 1.

```
def config(s)
  puts s
  if block_given? then
    yield
  end
end
```

Figure 5: Ruby program to display the value of the config element

```
['before','buy','at'].each do |m|
  Object.module_eval \%{
    def #{m}(s)
      puts s
      if block_given? then
        yield
      end
    end}
end
```

Figure 6: Ruby program to display three elements

Dynamic features in Ruby are helpful if we need to display the *before*, *buy*, and *at* elements shown in Figure 1. Figure 6 shows that a Ruby program to define three methods at run-time using closures and the "module_eval" method. This style is important in the design of a program generator (called "ribgen") to generate Ruby programs from the Ribbon definitions.

```
java_package 'test.ribbon'
java_prefix 'Rbn'
generate_java
generate_ribsetup 'setup_file.rb'
generate_diagram 'my_diagram.dot'
```

Figure 7: Specification to generate Ruby and Java programs

Ribgen generates a Ruby program, Java programs, and a class diagram for the generated Java classes. Figure 7 shows the specification as follows;

java_package specifies the path "test.ribbon" for the Java package

java_prefix specifies addition of a prefix "Rbn" at the beginning of the Java class name

generate_java specifies generation of Java classes

generate_ribsetup specifies generation of Ribbon-related programs to the file name "setup_file.rb"

generate_diagram specifies generation of a class diagram for generated Java classes to the file name "my_diagram.dot"

Ribbon is designed to map the representation to Java classes. Figure 8 shows a class diagram[3]for Java classes generated from Ribbon definitions in Figure 4. The RbnBase class is a base class for all classes generated by ribgen; it provides two fields: *name* and *value*. In Figure 8, there are five subclasses of RbnBase (RbnConfig, RbnBefore, RbnBuy, RbnAt, and RbnOrder), corresponding to the five elements (*config*, *before*, *buy*, *at*, and *order*).

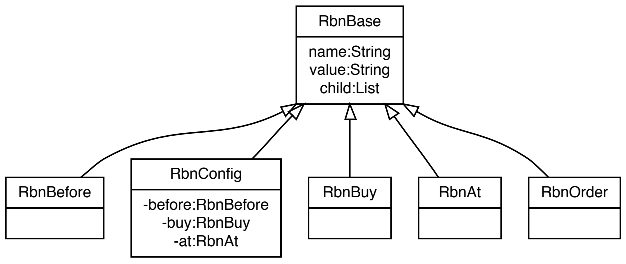

Figure 8: Class diagram of Java classes generated by ribgen

If the Ruby programs shown in Figure 6 are customized, we can develop a program to construct Java objects from the representation shown in Figure 1. Figure 9 shows Java objects corresponding to the Ribbon elements. The Ribbon representation and related programs are executed on a Ruby interpreter. The Ruby programs trigger instantiation one after another and instantiate all Java objects. After the construction of Java objects, the *RbnConfig* object has links to the *RbnBefore*, *RbnBuy*, and *RbnAt* objects. Moreover, the *RbnConfig* object has a list to access all child elements. The list is useful in developing programs that can traverse all child elements.

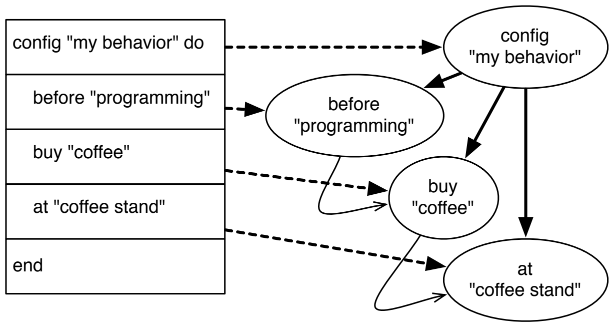

Figure 9: Relationships between Ribbon elements and Java objects

The RbnApi class provides some basic methods for developers in building Java programs using the Ribbon representation. Typical methods are:

RbnApi() is a constructor that initializes Ribbon APIs

RbnBase readRbnFile(String fileName) reads a Ribbon representation from the specified file and executes it on the Ruby interpreter.

void writeRbnFile(RbnBase obj, String fileName) writes the specified object (and objects reachable from it) in the specified file

All classes generated by ribgen are subclasses of RbnBase class. The typical methods of the classes are:

String getName() is an accessor to get the name of the element.

String getValue() is an accessor to get the value of the element

void setValue(String value) is an accessor to set the value of the element

void addChildFront(RbnBase obj) is a method to add the element to the front of the child elements

void addChildRear(RbnBase obj) is a method to add the element to the end of the child elements

java.util.List<RbnBase> getChild() is a method to get the list of child elements

Figure 10 shows a snippet of a Java program using Ribbon APIs. It reads a representation from a file "test-in.rb" and instantiates Java objects, after which the program
gets a RbnBefore object,
displays the RbnBefore object,
sets "A new configuration" to the RbnConfig,
gets a RbnAt object,
displays the RbnAt object, and
writes all objects to another file "test-out.rb."

```
RbnBase obj;
RbnConfig c;
RbnBefore b;
RbnAt a;
RbnApi rbnapi = new RbnApi();
obj = rbnapi.readRbnFile("test-in.rb");
c = (RbnConfig) obj;
b = c.getBefore();
System.out.println(b);
c.setValue("A new configuration");
ra = c.getAt();
System.out.println(a);
rbnapi.writeRbnFile(c,"test-out.rb");
```

Figure 10: An example of usage of Ribbon APIs

[3]The class diagram is generated by ribgen, and the methods are intentionally omitted from the diagram.

3.4 Current Implementation

The implementation work was performed on an Apple MacBook Pro with Mac OS X 10.4.11, JRuby 1.0.2, and Java 1.5.0_13. The program generator ribgen is written in Ruby. It reads a Ribbon definition file, generates a Ruby program to set up, and generates Java classes corresponding to all elements. Moreover, it generates a class diagram to understand all generated classes and the inheritance hierarchy. The class diagram in Figure 8 is an example of one generated from the Ribbon definitions shown in Figure 4 and 7.

The Ribbon representation changes to graph-structured data using unique identifiers and references. When a Java program, using Ribbon APIs, writes graph-structured objects to a file, the objects with cyclic paths need only be written once. Ribbon APIs correctly serialize them using the algorithm mentioned in the paper[17].

4 Summary

This paper described Ribbon, a new representation written in a text-based data format using Ruby syntax. One of its important features is that the representation is executable. Ribbon is useful for Java programs to read/write Java objects to persistent storage media, or to traverse the structured data.

A program generator was developed to create Ruby and Java programs from Ribbon definitions. In the author's experience, productivity was improved in the design and implementation of programs that manipulate structured data.

Ribbon and its related tools are now being used in applications supporting the development of commercial products, such as a diagram editor and a compiler front-end. The development and results will be published in a future paper.

References:

[1] Alfred V. Aho, Monica S. Lam et al., *Compilers : Principles, Techniques, and Tools, 2nd ed.* Pearson Education, 2006.

[2] Gernard Goos, William A. Wulf et al., *DIANA: An Intermediate Language for Ada*, Springer-Verlag, 1983.

[3] Object Management Group, OMG IDL, http://www.omg.org/gettingstarted/omg_idl.htm

[4] Richard Snodgrass, *The Interface Description Language: Definition and Use*, Computer Science Press, 1989.

[5] Sun Microsystems, Object Serialization, http://java.sun.com/javase/6/docs/technotes /guides/serialization/.

[6] Greg Badros, JavaML: A Markup Language for Java Source Code, In *9th International World Wide Web Conference*, http://www9.org/w9cdrom/index.html, 2000.

[7] Gregory McArthur, John Mylopoulos and Siu Kee Keith Ng, An Extensible Tool for Source Code Representation Using XML, In *9th Working Conference on Reverse Engineering*, pp.199–209, 2002.

[8] Katsuhisa Maruyama and Shinichiro Yamamoto, A CASE Tool Platform Using an XML Representation of Java Source Code, In *4th IEEE International Workshop on Source Code Analysis and Manipulation*, pp.158–167, 2004.

[9] Jonathan I. Maletic, Michael Collard and Huzefa Kagdi, Leveraging XML Technologies in Developing Program Analysis Tools, In *4th International Workshop on Adoption-Centric Software Engineering*, pp.80–85, 2004.

[10] Marjan Mernik, Jan Heering and Anthony M. Sloane, When and How to Develop Domain-Specific Languages, *ACM Computing Surveys*, Vol.37, No.4, pp.316–344, 2005.

[11] Paul Hudak, Building Domain-Specific Embedded Languages, *ACM Computing Surveys*, Vol.28, No.4es, p.196, 1996.

[12] Martin Fowler, MF Bloki: Domain Specific Language, http://martinfowler.com/bliki /DomainSpecificLanguage.html

[13] Ruby Programming Language, http://www.ruby-lang.org/en/.

[14] Evan Phoenix, Rubinius : The Ruby Virtual Machine, http://rubini.us/.

[15] JRuby - Home, http://jruby.codehaus.org/.

[16] IronRuby, http://www.ironruby.net/.

[17] Andrew Birrell, Greg Nelson et al., Network Objects, In *14th ACM Symposium on Operating Systems Principles*, pp.217-230, 1993.