Structured Data Representation for Multiple Programming Languages on Java VM

Kazuaki Maeda

***Abstract*—**This paper describes RugsOn, a new representation written in a text-based data format. The design principle of RugsOn is good readability and simplicity of structured data representation. One feature of RugsOn is a domain specific language to represent structured data. It is similar as JSON, but syntax of RugsOn is carefully chosen for multiple programming languages. Another feature is an executable representation. Once RugsOn-related definitions are loaded into a runtime environment, the representation can be executed corresponding to the definitions. Utility programs are available to read/write structured data to persistent storage-media, or to traverse the structured data. A program generator was developed to create Ruby, Groovy, Scala and Java programs from RugsOn 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

I. Introduction

THERE are many applications (e.g. graph drawing and editing tools) which have their own representation for structured data[1][2][3]. The representation has typically a textual format, which is easy to read and write. If the structured data can be defined in a language, a validation tool using the definition is very useful to find human errors in entering data manually. The definition is often modified during designing the structured data. At that time, the validation tool is effective to find design errors.

XML is mostly used as a standard language for the data representation in many application fields. XML can be used across different platforms, i.e., different computers, operating systems, and programming languages. To manipulate the representation in XML, according to the hierarchical tree structure, it is parsed to a hierarchical tree of elements and other XML entities in the main memory. After construction of the tree, each node can be accessed using tree traversal APIs. XML Schema provides a means to define the representation in XML. After defining the representation in XML Schema, we can check whether it conforms to the schema or not using a validation tool for XML. The schema and the representation in XML are written in human readable text format, but they are composed of many redundant tags, those are start tags and end tags, so that there are some cases where it is difficult to read and understand them.

JSON is one of popular representations for structured data [4][5]. The structured data in JSON is represented by object notation of JavaScript and it is easier to read and write than the representation in XML. A draft of JSON schema is now open for discussion [6]. It is a JSON-based format to define the representation in JSON. Some software tools including validation tools are being implemented. According to the draft, however, we have to use link description objects to define graph structured data in JSON. It is not easy task to define the graph structured data containing many of kinds of nodes for the graph editor.

From these considerations, we can obtain the following:

1. The representation of structured data should be simple and easy to read/write manually
2. A data definition language for structured data is needed and should be simple and easy to read/write

This paper describes RugsOn (which stands for Ruby, Groovy and Scala becoming Object notation), a new representation of structured data in a text-based data format using multiple languages, Ruby, Groovy and Scala, which are available on Java VM. It was designed to satisfy two important points as mentioned above, and to develop the commercial software. Design principles of RugsOn are the following;

- RugsOn was designed to reach good readability and simplicity of structured data representation
- A data definition language was designed using a subset of Ruby syntax so that we can define structured data using symbols, strings and methods in Ruby

RugsOn and its related tools currently support the multiple programming languages. An important feature of RugsOn is that the representation is executable. For example, if a representation in RugsOn is successfully loaded into a Ruby interpreter, the parsing of the data is already done; we therefore do not need to parse the representation in RugsOn. If RugsOn related classes and methods are loaded into the Ruby interpreter, the data representation can be executed corresponding to the definitions. It is useful in reconstructing objects from the representation or for traversing the structured data.

In this paper, Section 2 describes related works about structured data. Section 3 explains RugsOn and its definitions. Section 4 summarizes this paper.

Kazuaki Maeda is with the Department of Business Administration and Information Science, Chubu University, Kasugai, Aichi, 487-8501 Japan (corresponding author to provide phone: +81-568-51-1111; fax: +81-568-52-1505; e-mail: kaz@acm.org).

II. Structured data and Domain Specific Languages

A variety of representations for structured data have been developed to date. As a practical application, we can use an interface description language, IDL (The IDL in this context is different from OMG IDL[7]) compiler in a Scorpion toolkit[8], which was developed more than twenty years ago. IDL is used to define structured data; the IDL compiler reads the specification in the IDL and generates useful functions to manipulate structured data in the C programming language, including a writer to translate the structured data 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[9], 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 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. 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.

Domain-specific language (DSL) is a computer language of limited expressiveness focused on a particular domain. Martin Fowler's book[10] describes an external DSL and an internal DSL. In the book, they are defined as the following:

- An external DSL is a language separate from the main language of the application it works with.
- An internal DSL is a particular way of using a general-purpose language.

External DSLs have their own custom syntax. The examples include SQL, Yacc, and XML configuration files. In the applications with the external DSLs, we need parsers using text parsing techniques. XML syntax is completely different from general-purpose languages such as Java, C++ and Ruby, but XML is frequently chosen as an external DSL to describe structured data.

JSON is an internal DSL in developing JavaScript programs. The structured data in JSON is represented by object notation of JavaScript so that we do not need to prepare a parser to read it. All we have to do is evaluate it. It is easy to build programs to process the representation in JSON and it gives a simple way to program if we choose JavaScript as a host language. However, if we choose different programming languages from JavaScript, such as Java and C++, the representation in JSON is parsed to a hierarchical tree for manipulation in the main memory and each node can be accessed using tree traversal APIs. It is similar as an external DSL like XML.

The author believes the following;

- Structured data should be represented using an internal DSL, and
- It should be available for multiple programming languages.

Therefore, a common subset of Ruby, Groovy and Scala was carefully chosen for RugsOn design so that one representation in RugsOn is available for multiple programming languages.

III. Representation and Definition of Structured Data

A. RugsOn as an Object Notation

RugsOn is used to represent graph structured data. The representation in RugsOn is composed of several elements. Each element has a value and a name. For example,

```
"www" .nodeName
```

which represents the value as "www" and the name of the element as *nodeName*. The element is not only a data representation, but also it is an executable method invocation without parentheses in programming languages (i.e. Ruby, Groovy and Scala). The method name is *nodeName* and the receiver of the method is "www."

In RugsOn, a structure is represented using a block as an argument. For example, Fig. 1 shows that *node* has four child elements: *nodeName*, *kind*, *size* and *_id*. In the case of the first *node* element, *nodeName* element's value is "www," the *kind* element's value is 1, the *size* element's value is 0.7, and *_id* element's value is "`id0`."

An element to represent a collection can have more than one element with the same name. In Fig. 1, the *nodes* element has three child elements with the same name *node*. The first *node*

```
cloneGraph {
 nodes {
  node {
   "www".nodeName
   1    .kind
   0.7  .size
   "id0"._id
  }
  node {
   "isem".nodeName
   1    .kind
   0.7  .size
   "id1"._id
  }
  node {
   "org".nodeName
   1    .kind
   0.7  .size
   "id2"._id
  }
 }
```

```
 links {
  link {
   ":id0".source
   ":id1".target
   2      .kind
   true   .visible
  }
  link {
   ":id1".source
   ":id2".target
   2      .kind
   true   .visible
  }
  link {
   ":id2".source
   ":id0".target
   3      .kind
   false  .visible
  }
 }
}
```

Fig. 1 Graph-structured data in RugsOn

element has a *nodeName* element with a value "www," the second *node* element has a *nodeName* element with a value "isem," and the third *node* element has a *nodeName* element with a value "org." This represents a sequence of *node* elements. If we need to represent an element linked to another element across the structure, a unique identifier, called *_id*, is given to the element, and another element refers to the element using the identifier. In Fig. 1, a *_id* element with the identifier "`id0`" is given to the first *node* element, and another *_id* element with the identifier "`id1`" is given to the second *node* element. The first character ':' in the string constant, for example "`:id0`," means a reference to another element. The *source* element in the first *link* element has a reference to the identifier "`id0`," and the *target* element in the first *link* element has a reference to another identifier "`id1`." The representation shows links across the structure so that RugsOn supports graph structured data. As a result, the representation in Fig. 1 logically means the graph shown in Fig. 2. If the value of the *visible* element is `true`, the solid line is drawn in the figure. If the value of the *visible* element is `false`, the dotted line is drawn.

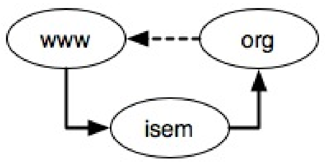

Fig. 2 Logical meaning of the graph representation

When a program writes graph structured data to a file and another program reads the data from the file, it reconstructs the graph structured data. RugsOn specific APIs are prepared so that the value of the *_id* element can be freely modified.

B. Definition of Structured Data

Structured data is defined using symbols in Ruby and the six keywords as follows:

- *has* : composition of elements
- *is_a* : specialization of an element
- *is_seq_of* : sequence of elements
- *is_type* : primitive data type (byte, char, short, int, long, float, double, boolean, and string)
- *is_alias_of* : element name different from the type name
- *imports* : usage of external type, e.g., java.util.ArrayList as nodeList

Fig. 3 shows the definitions of the representation in Fig. 1. The definitions are shown as follows:

- *cloneGraph* element has two child elements: *nodes* and *links*
- *nodes* element is a collection of *node* elements
- *links* element is a collection of *link* elements
- *node* element has three child elements: *nodeName*, *kind*, and *size*
- *nodeName* element has a string value
- *kind* element has an integer value
- *size* element has a double value
- *link* element has four child elements: *source*, *target*, *kind*, and *visible*
- *source* element is a node type
- *target* element is a node type
- *visible* element has a boolean value

The author has considerable experience developing parsers from scratch to analyze source code. Recently, the source code analyzer for Java version 5.0 was developed. The analyzer will be ready for use in commercial products. The task has been complicated because approximately 200 classes, containing more than 1,000 methods and fields, were required to build an AST (abstract syntax tree) for Java. Ensuring their correctness and consistency in naming the definitions of classes, methods, and fields was tedious and error-prone.

The "convention over configuration" concept has been popularized since Ruby on Rails has been widely used for development of web applications[11]. An important point is to reduce the number of decisions so that developers are freed from the complicated work of configuration. On the basis of my experiences in designing and implementing many ASTs, the author notices that the names of fields, classes, and methods tend to be related. For example, to define a Java class for a unary expression, the class can contain

- a field `exp` of the type `ClsExp` to hold an operand,
- a field `opr` to hold an operator,
- a getter method `getExp()` to get a `ClsExp` object from the field `exp`,
- a setter method `setExp()` to set a `ClsExp` object to the field `exp`, and
- a method `access()` for Visitor design pattern

with documentation comments for the Javadoc utility. Moreover, a collection of expressions, `ClsExps`, may be needed to represent a list of `ClsExp`, and an `Iterator` class

```
:cloneGraph.has :nodes,:links
:nodes.is_seq_of :node
:links.is_seq_of :link

:node.has :nodeName,:kind,:size
:nodeName.is_type :string
:kind.is_type :int
:size.is_type   :double
:link.has :source,:target,:kind,:visible
:source.is_alias_of :node
:target.is_alias_of :node
:visible.is_type :boolean

:java.with_prefix "Grph"
:java.is_generated_in "org.graph1"
:ruby.is_generated_in "org.graph2"
:groovy.is_generated_in "org.graph3"
:scala.is_generated_in "org.graph4"
:diagram.is_generated "grphdiag.dot"
```

Fig. 3 Example of RugsOn definitions

is needed to traverse all objects in the unary expression.

Providing a software tool that generates the classes from a definition of the unary expression can decrease the cost of developing the classes. The tool should generate templates for most of the classes (`ClsExp`, `ClsExps`) and the methods (`getExp()`, `setExp()`) from the name `exp` only. For this study, the program generator RugsGen was implemented as the software tool shown in Fig. 4. It is very useful for developing programs from the RugsOn definitions.

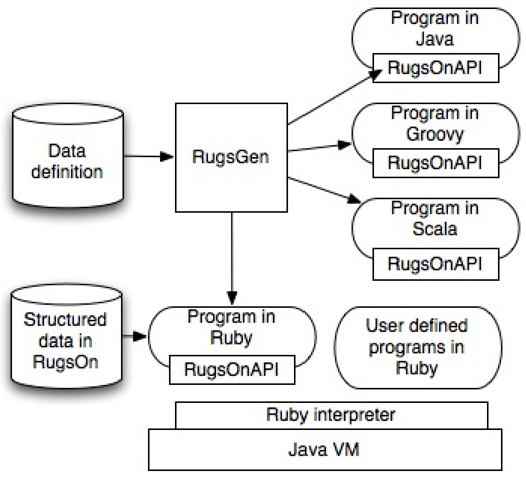

Fig. 4 Programs generated by RugsGen and the applications on Java VM

C. Program Generation for Multiple Languages

RugsGen generates Ruby programs, Groovy programs, Scala programs, Java programs, and a class diagram for the generated classes. In Fig. 3, some keywords are used as the following;

- *:java.with_prefix* specifies addition of a prefix "Grph" at the beginning of the Java class name
- *:java.is_generated_in* specifies generation of Java programs with the package path "org.graph1"
- *:ruby.is_generated* specifies generation of RugsOn-related Ruby programs under the directory "org.graph2"
- *:groovy.is_generated_in* specifies generation of RugsOn-related Groovy programs with the package path "org.graph3"
- :*scala.is_generated* specifies generation of RugsOn-related Scala programs with the package path "org.graph4"
- *:diagram.is_generated_in* specifies generation of a class diagram for generated Java classes to the file name "grphdiag.dot"

The representation in RugsOn becomes executable if RugsOn-related programs are loaded into the runtime environment. For example, Fig. 5 shows a snippet of a Ruby program to display the value of the *nodeName* element, the *kind* element, the *size* element and the *_id* element in Fig. 1. In the `RugsOnBuilder` class, the six lines are written in RugsOn,

```
class RugsOnVisitor
  def visit_node()
    puts "visit_node"
  end
  def leave_node()
    puts "leave_node"
  end
end
class RugsOnAcceptor
  @@visitor = RugsOnVisitor.new
  def self.node()
    @@visitor.visit_node()
    if block_given?
      yield
    end
    @@visitor.leave_node()
  end
end
class String
  def nodeName()
    puts self
  end
  def _id()
    puts self
  end
end
class RugsOnBuilder < RugsOnAcceptor
  node {
    "www".nodeName
    1   .kind
    0.7 .size
    "id0"._id
  }
end
```

Fig. 5 Snippet of Ruby programs generated and customized by RugsGen

and they are executable.

RugsGen generates a class diagram such as Fig. 6. Rectangles with light-grey color show predefined RugsOn-related classes and other rectangles show the classes defined in Fig. 3. RugsOn is designed to map the representation to Java classes. The `GrphBase` class is the base class for all classes generated by RugsGen; it is specified in Fig. 3 and provides three fields: *value*, *name* and *_id*. The RugsOn representation and related programs are executed on the runtime environment. For Java programs, the Ruby, Groovy and Scala programs can trigger instantiation one after another and instantiate all Java objects. After the construction of Java objects, the `GrphCloneGraph` object has links to the `GrphNodes` and `GrphLinks` objects. Moreover, the `GrphNodes` object has a list to access all child elements.

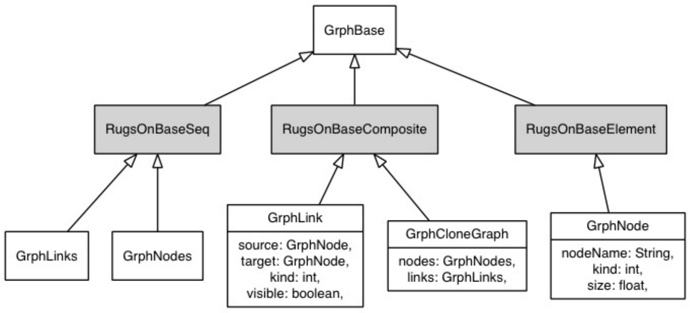

Fig. 6 Class diagram generated by RugsGen

D.Current Implementation

The implementation work is now going on an Apple MacBook Pro with Mac OS X 10.6.8, JRuby 1.6.0, Groovy 1.7.4, Scala 2.8.1, and Java 1.6.0_26. The program generator RugsGen is written in Ruby. It reads RugsOn definition files, and generates programs in multiple programming languages corresponding to all elements. Moreover, it generates a class diagram in Dot[1] to understand all generated classes and the inheritance hierarchy.

The structured data in RugsOn is stored in a database. Programs with RugsOn communicate RugsOnServer via RugsOnAPIs shown as Fig. 7. If scala is chosen for a programming language, the structured data in RugsOn is compiled and the class files generated by the scala compiler are also stored in RugsOnServer.

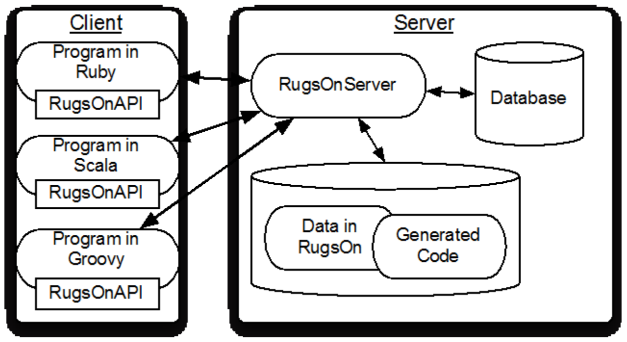

Fig. 7 RugsOnServer to store structured data in RugsOn

RugsOn supports graph data structures using unique identifiers and references. When a Java program, using RugsOn APIs, writes graph structured data to a file, objects with cyclic paths should be written only once. RugsOn APIs correctly serialize it using a hash table based on the algorithm mentioned in the paper[12].

IV. CONCLUSION

This paper described RugsOn, a new representation written in a text-based data format for multiple programming languages. One of its important features is that the representation is executable. RugsOn is available for multiple programming languages on Java VM to read/write objects to persistent storage media, or to traverse the structured data.

A program generator was developed to create Ruby, Groovy, Scala and Java programs from RugsOn definitions. In the author's experience, productivity was improved in the design and implementation of programs that manipulate structured data. RugsOn 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] E. Koutsofios and S. C. North, "Drawing Graphs with dot - dot User's Manual," Technical Report, AT&T Bell Laboratories, Murray Hill, New Jersey, 1993.
[2] Georg Sander, VCG Visualization of Compiler Graphs, User Documentation, 1995.
[3] The GML File Format, http://www.infosun.fim.uni-passau.de/Graphlet/GML/.
[4] JSON, http://www.json.org/.
[5] The application/json Media Type for JavaScript Object Notation (JSON), RFC 4627, http://www.ietf.org/rfc/rfc4627.txt , 2006.
[6] A JSON Media Type for Describing the Structure and Meaning of JSON Documents ,draft-zyp-json-schema-01, http://tools.ietf.org/html/draft-zyp-json-schema-01, 2009.
[7] Object Management Group, OMG IDL, http://www.omg.org/gettingstarted/omg_idl.htm
[8] Richard Snodgrass, The Interface Description Language: Definition and Use, Computer Science Press, 1989.
[9] Sun Microsystems, Object Serialization, http://java.sun.com/javase/6/docs/technotes/guides/serialization/.
[10] Martin Fowler, Domain-Specific Languages, Addison-Wesley, 2011.
[11] D. Thomas, D. H. Hansson, et al., Agile Web Development with Rails, The Pragmatic Programmers, 2nd edition, 2006.
[12] Andrew Birrell, Greg Nelson, et al., Network Objects, 14th ACM Symposium on Operating Systems Principles, pp.217-230, 1993.

Kazuaki Maeda is a professor of Department of Business Administration and Information Science at Chubu University in Japan. He is a member of ACM, IEEE, IPSJ and IEICE.

He graduated in Department of Administration Engineering from Keio University at 1985, and graduated in Graduate School of Science and Technology from Keio University at 1990. After leaving Keio University, he got an academic position at Chubu University. His research interests are Compiler Construction, Domain Specific Languages, Object-Oriented Programming, Software Engineering and Open Source Software.