Declarative Language for SAX Handler Definition

Alexey Vladykin
St. Petersburg State University of Information Technologies, Mechanics and Optics
vladykin@gmail.com

Abstract

In this paper a declarative language for SAX handler definition is proposed. This language allows to describe complex XML parsing algorithms in a simple manner. An algorithm is introduced for automatic transformation of such handler descriptions into finite state machines, and then into source code. This approach reduces the complexity of SAX handler development by eliminating the greater part of error-prone manual work.

1. Introduction

XML is widely used for a variety of purposes: from storing program configuration to transmitting data packets over the Internet. More and more applications require XML support, and programmers often have to develop code that extracts data from XML documents.

Some XML documents are very large: up to hundreds of gigabytes. E. g. a complete Wikipedia dump including all articles with their change history takes 148 Gb (bzip2-compressed).

How can one parse such document and extract some useful information from it? The only feasible approach for documents like Wikipedia dump is Simple API for XML (SAX) [3], because SAX parser is very effective about computer resources and passes XML content to the rest of application in small portions. Other well-known approaches to XML parsing like Document Object Model (DOM) [6] and Java API for XML Binding (JAXB) [1] need to load the whole document into RAM, which is unacceptable.

The major SAX drawback is complexity of crafting SAX handlers manually. [4] This paper describes an approach which significantly simplifies development of SAX handlers. A declarative language for XML handler definition is introduced. Handler definition in this special language is automatically translated into finite state machine, and then into source code in any programming language.

2. Simple API for XML

SAX is a low-level XML parsing technique. SAX parser sequentially reads input document and notifies SAX handler about every start and end tag, as well as about character data between tags.

SAX parser implementations exist for many programming languages as part of standard library or as a 3rd party library.

SAX handler must be written by hand for each document type that needs to be processed. Crafting SAX handlers may be very hard task in case of complex document structure.

Handler class in Java extends class *DefaultHandler* and typically overrides the following three methods with hand-written code:

void startElement(String uri, String localName, String qName, Attributes attrs)

void endElement(String uri, String localName, String qName)

void characters(char[] ch, int start, int length)

Handler calls *startElement* method when it encounters start tag, *endElement* is called for end tag, and *characters* – for character data between tags. [2]

For other languages SAX handler structure is essentially the same.

3. SAX Handler as an Entity with Complex Behavior

It is important to understand the reasons why writing SAX handlers is so complex, and to eliminate those reasons. This task can be accomplished with automata based approach. [7]

SAX handler receives notifications from SAX parser and translates those notifications into commands or data structures that can be understood by the rest of the application. It has to track current parser position within the document and, according to that position, handle notifications differently.

SAX handler is an entity with complex behavior [7], because its reaction to incoming notifications depends on previously received notifications. Handler can be logically divided in two parts:

a) **controlling part**, which tracks current parser position in XML document by remembering the history of incoming notifications, and chooses one of the possible commands for controlled part;

b) **controlled part**, which receives commands from controlling part and translates them into commands or data structures for the rest of the application.

The biggest challenge usually is controlling (behavioral) part, because tracking of current parser position in XML document within traditional approach requires setting and checking many boolean flags, or other tricks. Controlled part is typically simple, but its code is spread and lost inside controlling part.

According to the paradigm of automata based programming, an entity with complex behavior should be represented as an automated object. Explicit separation of controlling and controlled parts of SAX handler can help crafting such handlers and make their structure and behavior much clearer. Moreover, it is possible to generate the code of controlling part based on a declarative definition. The next section describes a language that can be used for declarative definition of SAX handlers.

4. Declarative Language for SAX Handler

We'll demonstrate the language and its usage on a simple problem of extracting a list of departments and all non-terminated employees from an XML document describing company structure.

Such document could look like this:

```
<company>
<name>Mr. X and Partners</name>
<department>
   <name>Human Resources</name>
   <employee>
      <name>John Smith</name>
   </employee>
   <!-- more employees -->
</department>
<department>
   <name>Engineering</name>
   <employee terminated=”true”>
      <name>Zzyzzy Zzyrryxxy</name>
   </employee>
   <!-- more employees -->
</department>
<!-- more departments -->
</company>
```

Document structure shown here is relatively simple, but it exposes a typical problem of distinguishing between *name* of a company, *name* of department and *name* of employee.

Another typical problem here is that extracting *John Smith* from fragment *<name>John Smith</name>* requires adding some lines of code to all three of SAX handler methods: *startElement*, *endElement* and *characters*. Thus a logically atomic extraction of employee name is split into several stages.

Both mentioned problems are related to the controlling part of SAX handler and imply saving handler state between invocations of its methods. This is what makes writing SAX handlers complicated.

Let's see how these problems can be solved when SAX handler is described using the proposed declarative language. Handler definition in the proposed language looks like:

```
(
<department>
   <name> { capture(); }
   </name> { obj.addDepartment(captured()); }
   (
   <employee terminated!=”true”>
      <name> { capture(); }
      </name> { obj.addEmployee(captured()); }
   </employee>
   )*
</department>
)*
```

This definition consists of the following elements:

a) start and end tags. Start tag may have constraints on its attributes (e.g. *terminated!=”true”*). Arbitrary code may be specified after tag in braces. It will be executed when such tag is encountered in document.

b) parentheses, used to group tags and specify zero-or-one (?) and zero-or-more (*) quantifiers, or enumerate alternatives separated by |.

Here is the formal grammar for this language:

```
S :: START_TAG | END_TAG | GROUP
START_TAG :: "<" ID OR_EXPR? ">" ACTION?
END_TAG :: "</" ID ">" ACTION?
GROUP :: "(" ( START_TAG | END_TAG
      | GROUP )* ( "*" | "?" )? ")"
ACTION :: "{" CODE "}"
OR_EXPR :: AND_EXPR ( "||" AND_EXPR )*
AND_EXPR :: TERM ( "&&" TERM )*
TERM :: ID "==" STRING | ID "!=" STRING
      | ID "=~" STRING | ID "!~" STRING
      | "(" OR_EXPR ")"
```

In this grammar *ID* is any valid tag or attribute name; *STRING* is arbitrary text enclosed in quotation

marks or keyword *null*; *CODE* is arbitrary code in target programming language.

According to the paradigm of automata based programming our SAX handler will consist of two parts: controlling part (automaton) and controlled part. Declarative definition of controlling part is shown above.

Controlled part provides some interface to controlling part. In our case this interface consists of two methods: *void addDepartment(String name)* and *void addEmployee(String name)*. Implementation of these methods and the whole controlled part class is up to the developer.

Controlling automaton calls methods of controlled object according to the declarative definition shown above. Additionally it can use two utility methods provided by controlling automaton: *void capture()* and *String captured()*. The former tells automaton to capture character data coming from parser into temporary buffer. The latter returns character data captured since last call to *capture* and clears the buffer.

5. Building Automaton for SAX Handler

To build controlling automaton from its declarative definition we need to extract its states and build transitions between states.

Both tasks are rather simple, because states and transitions are implicitly present in the declarative definition. Informally speaking, each place between tags corresponds to one state, and tags correspond to transitions.

There are some corner cases related to parentheses and quantifiers, but overall algorithm is pretty straightforward. Its implementation in Java can be found at project website [5].

6. Code generation

Given the description of SAX handler controlling automaton as a set of states and transitions, it is possible to automatically generate source code in virtually any programming language. Currently implemented is code generation for Java. [5]

Code generator creates a class that extends *DefaultHandler* and overrides aforementioned methods *startElement*, *endElement* and *characters*. All transitions on start tags are placed in *startElement*, and all transitions on end tags – in *endElement*.

Automaton has only two member variables: current state (integer) and temporary buffer for character data. Thus memory consumption is minimal.

Every call to automaton's *startElement*, *endElement* or *characters* is a quick constant time operation (not taking into account what happens in invoked methods of controlled object), so parsing XML document with automated SAX handler remains an efficient linear algorithm.

7. Conclusion

In this paper a declarative language for SAX handler definition is introduced, and an automatic code generation system is described. The system takes declarative definition of SAX handler as input, builds controlling automaton and then translates it into source code in some programming language. Currently code generation is implemented for Java programming language. Implementation of the remaining part of SAX handler – controlled object – is to be written manually by the programmer, but this part is typically trivial. Proposed approach features significant simplification of SAX handler creation, and does not bring any performance penalty.

This approach has been used in development of an application that extracted contents of several kinds of complex table-like structures from thousands of XML documents.

8. References

[1] JAXB Reference Implementation. https://jaxb.dev.java.net/

[2] McLaughlin B. *Java and XML, Second Edition*. O'Reilly, 2001.

[3] Official website for SAX. http://www.saxproject.org/

[4] Oleg Kiselyov. “A better XML parser through functional programming”. *LCNS*, Springer-Verlag, 2002, pp. 209-224.

[5] SaxGen – Google Code. http://code.google.com/p/saxgen/

[6] W3C Document Object Model. http://www.w3.org/DOM/

[7] Поликарпова Н. И., Шалыто А. А. *Автоматное программирование*, Питер, СПб., 2009.