A CASE Tool Platform Using an XML Representation of Java Source Code

Katsuhisa Maruyama[†]
Department of Computer Science
Ritsumeikan University
1-1-1 Noji-higashi Kusatsu
Shiga 525-8577, Japan
maru@cs.ritsumei.ac.jp

Shinichiro Yamamoto
Department of Information Systems
Aichi Prefectural University
1522-3 Ibaragabasama Kumabari Nagakute-cho
Aichi-gun Aichi 480-1198, Japan
yamamoto@ist.aichi-pu.ac.jp

Abstract

Recent IDEs have become more extensible tool platforms but do not concern themselves with how tools running on them collaborate with each other. They compel tool developers to use proprietary representations or the classical AST to build software tools managing source code. Although these representations contain sufficient information, they are neither portable nor extensible. This paper proposes a tool platform that manages commonly used, fined-grained information about Java source code by using an XML representation. Our representation is suitable for developing tools which browse and manipulate actual code since the original code is both structured with tags and stored in textual elements of a converted XML document. Additionally, it exposes information resulting from global semantic analysis, which is never provided by the typical AST. Our proposed platform allows the developers to extend the representation for the purpose of sharing or exchanging various kinds of information about source code, and also enables them to build new tools by using existing XML utilities.

1. Introduction

Object-oriented software is hard to develop without integrated development environments (IDEs) since it contains many classes and the relationship between them. A remarkable point is that a recently released IDE is not only a collection of programming tools but also an extensible tool platform. For example, Eclipse [2] has a powerful plug-in mechanism for easily adding a new tool to and removing an existing tool from itself.

By supporting the plug-in mechanism, developers have chances to build their own tools and desire their tools to collaborate with each other. Accordingly, a tool platform must collect the detailed information about programs being developed and then present it in proper form that can meet diverse developers' requirements. Unfortunately, the conventional tool platforms store information about source code by using either proprietary representations or the typical abstract syntax tree (AST) [13]. Of course these representations contain sufficient information and several powerful tool platforms such as Eclipse [2], the DMS Software Reengineering Toolkit [7], or RECORDER [6] provide well-designed application programming interfaces (APIs) for accessing the information. However, the classical representations are neither portable nor extensible. That is, none of the conventional platforms concern themselves with how a newly built tool stores additional information obtained through its execution and exchanges such information with other tools. In addition, only the prepared APIs are insufficient for building diverse tools. Therefore, the tool developers tend to create overhead modules, which are used for extracting necessary information from the integrated representation, in their respective tools, or might have to modify integrated modules and date structure. To build various kinds of software tools managing source code and make them collaboratively work without much effort, a tool platform should use a not only simply standard but also portable and extensible representation, which is the medium for sharing and exchanging source-code information and allows the developers to add individual information they define.

The authors have developed a tool platform with a software repository that can store and provide fine-grained information about Java source code by using the extensible markup language (XML) [3]. This paper proposes this tool platform and a new XML-based representation, which are called Sapid/XML (sophisticated APIs for CASE tool development with an XML repository) and XSDML (extensible software document markup language), respectively.

[†] He is also with Institute for Software Research, University of California, Irvine. Irvine, CA 92697-3425 USA.

In XSDML documents converted from source code, code fragments are classified by marking with respective tags and structured by nesting the tags based on the structure of the source code. Additionally, these documents contain additional information resulting from syntactic and semantic analysis. In respect of portability, the motivation of Sapid/XML (or XSDML) is analogous to that of conventional XML representations such as a markup language for Java source code (JavaML) [14] and an XML-based representation for object-oriented source code (OOML) [19]. However, its target differs much from their targets. The purpose of Sapid/XML is to facilitate developers building tools which manipulate actual source code. Sapid/XML provides a fine-grained representation which is an alternative to the classical AST containing all information about source code while JavaML provides a highly abstract representation of source code. Moreover, Sapid/XML inserts several useful links obtained through global semantic analysis for the whole of source code, which are not supported by JavaML and OOML.

Sapid/XML both makes Java source code more portable and convenient since our proposed XSDML is based on XML which is a simple, widely used text-based format and the XSDML exposes the structure and relationship lurking in source code. Many existing XML utilities can be used for examining and manipulating the source code. Sapid/XML also allows developers to extend the prepared representation. They can define new tags and attributes to share common information and exchange specific one although its definition needs a simple consistency check for the document type definition (DTD) [3]. It is useful for building new software tools to extend a representation of source code without examining and modifying modules in a tool platform. Here we have to mention that Sapid/XML does not strive to dismiss existing IDEs. It shows the potential of a tool platform using an XML representation of source code.

We first present an overview of Sapid/XML and explain how Java source code is converted to an XSDML document and how software tools access the converted XSDML documents. Next we show several software tools running on Sapid/XML. Then we give experimental results in respect to the performance of Sapid/XML. Finally, we conclude with a summary.

2. Sapid/XML Tool Platform

Sapid/XML generates XML documents represented in our proposed XSDML from Java programs (written in Java 1.4) and provides them for software tools. Figure 1 shows an overview of the Sapid/XML tool platform. It mainly consists of four components: a source code converter (a syntactic parser and a semantic analyzer), access libraries, a Java-XML software repository, and Java wrappers. This section explains how Java programs are converted into XSDML documents and what information is contained in the documents, and describes access libraries and Java wrappers accessing these documents.

Table 1. Elements of the XSDML

Element name	Fragment of Java source code
`File`	compilation unit (File)
`Package`	package declaration
`Import`	import declaration
`Class`	class declaration
`Intf`	interface declaration
`SInit`	static initializer
`Ctor`	constructor declaration
`Method`	method declaration
`Field`	field declaration
`Param`	formal parameter
`Local`	variable declaration
`ExtdOpt`	superclass clause
`ImplOpt`	superinterface clause
`ThrwOpt`	throws clause
`Members`	class/interface body
`Qname`	qualified identifier
`Type`	type
`Stmt`	statement
`Label`	label declaration
`Expr`	expression
`ident`	identifier
`literal`	literal
`comment`	comment
`kw`	Keyword
`op`	operator
`sp`	blank or tab character
`nl`	new line character

2.1. Syntactic Parser

The XSDML represents source code as twenty non-terminal and seven terminal elements, which are shown in Table 1. The terminal element has only the textual content while the non-terminal element can nest others. The syntactic parser inserts directly these elements into the original source code without changing the contents of the code, that is, it only adds tags and attributes in the original code. Each of the code fragments is delimited by the tags and all tokens (identifiers, keywords, comments, white spaces, and new lines) of the code remain in the textual contents of the terminal elements. The original source code can be restored from the converted XSDML document by removing all tags and leaving behind the textual contents of elements. Attributes are available to represent additional properties such as modifiers, accessibility settings, fully-qualified names, and sorts of elements. For example, types (`Type`), statements (`Stmt`), expressions (`Expr`), and literals (`literal`) elements are classified as three, fifteen, fifty-nine, and six by the attribute `sort`, respectively. Parts of the values of the `sort` attribute are listed in Table 2 [1].

[1] See http://www.jtool.org for details.

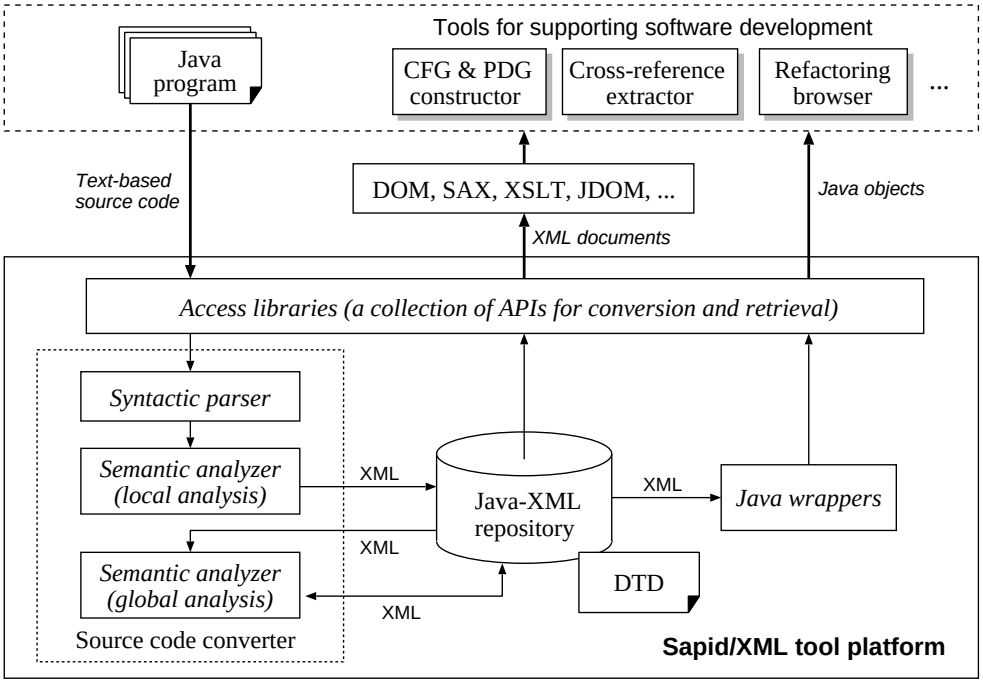

Figure 1. Overview of the proposed tool platform.

Table 2. Values of the sort attribute

sort of `Type`	**Description**
`Array`	array type (`[]`)
`Primitive`	primitive type (including void)
`Object`	reference type

sort of `Stmt`	**Description**
`EMPTY`	empty statement
`EXPR`	expression statement
`BLOCK`	block
`DOWHILE`	**do** statement
`WHILE`	**while** statement
`FOR`	**for** statement
`IFELSE`	**if-then** statement
`SWITCH`	**switch** statement
`BREAK`	**break** statement
`CONT`	**continue** statement
`RETURN`	**return** statement
`SYNC`	**synchronized** statement
`THROW`	**throw** statement
`TRY`	**try** statement
`ASSERT`	**assert** statement

sort of `Expr`	**Description**
`MUL`	multiplication operator (`*`)
`DIV`	division operator (`/`)
`MOD`	remainder operator (`%`)
`ADD`	addition operator (`+`)
`SUB`	subtraction operator (`-`)
`Assign`	assignment operator (`=`)
`Assign$op`	compound assignment operator, the $op is one of arithmetic operator (e.g., MUL)
`DOT`	dot operation (`.`)
`InstanceOf`	type comparison (`instanceof`)
`Plus`	unary plus operator (`+`)
`Minus`	unary minus operator (`-`)

sort of `Expr`	**Description**
`PreINC`	prefix increment operator (`++`)
`PreDEC`	prefix decrement operator (`--`)
`PostINC`	postfix increment operator (`++`)
`PostDEC`	postfix decrement operator (`--`)
`LogicalNOT`	logical complement operator (`!`)
`LogicalAND`	conditional-and operator (`&&`)
`LogicalOR`	conditional-or operator (`||`)
`CondLT`	comparison operator (`<`)
`CondGT`	comparison operator (`>`)
`CondLE`	comparison operator (`<=`)
`CondEQ`	equality operator (`==`)
`CondNE`	equality operator (`!=`)
`CondGE`	comparison operator (`>=`)
`CtorCall`	call to a constructor
`SpCtorCall`	call to a superclass's constructor
`MethodCall`	method call
`ArrayAccess`	array access expression
`InstanceCreation`	class instance creation expression
`ArrayCreation`	array creation expression
`Cast`	cast expression
`Paren`	parenthesized expression
`VarRef`	variable reference
`Literal`	literal
`This`	`this` operation
...	

sortof `literal`	**Description**
`INT`	integer literal
`FLOAT`	floating-point literal
`STR`	string literal
`CHAR`	character literal
`BOOL`	boolean literal
`NULL`	null literal

```
<?xml version="1.0"?>
<!DOCTYPE File SYSTEM "JX-model3-ext.dtd">
<File classpath="/usr/home/maru/Work/Report/scam04/xsdml-examples/FirstApplet" id="s792723457" pat..
</nl><Import id="d0"><kw>import</kw><sp> </sp><QName id="s843055105"><ident defid="s843055105">jav..
</nl><nl line="3" offset="41">
</nl><Class access="Public" fqn="FirstApplet" id="s796917761"><kw>public</kw><sp> </sp><kw>class</..
</nl><sp>  </sp><Method access="Public" id="s809500673" typefirst="s813694978"><kw>public</kw><sp>..
</nl><sp>    </sp><Stmt id="s826277890" sort="EXPR"><Expr id="s830472193" sort="DOT"><Expr id="s83..
</nl><sp>  </sp><op>}</op></Stmt></Method><nl line="7" offset="162">
</nl><op>}</op></Members><Ances distance="0" name="FirstApplet" sort="CLASS"></Ances><Ances distan..
</nl><FqnMap fqn="java.awt.MenuContainer" jar="rt.jar" path="java/awt/MenuContainer.class"></FqnMa..
```

Figure 3. Document represented in XSDML.

```
1: import java.applet.*;
2: import java.awt.*;
3:
4: public class FirstApplet extends Applet {
5:   public void paint(Graphics g) {
6:     g.drawString("FirstApplet", 25, 50);
7:   }
8: }
```

Figure 2. Java source code.

The simple source code quoted from [14] and an XSDML document converted from it are shown in Figure 2 and Figure 3, respectively. Each line of the document except the XML headers corresponds to that of Java source code since the XSDML retains every new line. Moreover, the original code can be seen in the textual contents of the terminal elements (e.g., blanks or keywords are enclosed with the `<sp>` or `<kw>` tag). This crude document is hard for human to read but we can use various XML utilities to view it. Figure 4 illustrates a tree view of the XSDML document shown in Figure 3 as displayed by the Mozilla [5].

Here it is worth discussing the problems Badros pointed out in [14]. He stated that the representation marked-up by only adding tags would need to further lexically analyze the contents of the elements and it would not sufficiently abstract the classical source code. To alleviate these problems, the XSDML introduces a fine-grained tagging and slightly verbose attributes. For example, the method call at line 6 in the source code shown in Figure 2 is converted into [1]:

```
<Stmt id="s826277890" sort="EXPR">
  <Expr id="s830472193" sort="DOT">
  <Expr id="s830472194" sort="VarRef"
        read="yes" write="yes">
    <ident defid="s805306369">g</ident>
  </Expr><op>.</op>
  <Expr id="s830472195" sort="MethodCall">
    <ident defid="c302" fqn="void"
           ref="java.awt.Graphics">
      drawString</ident>
    <op>(</op>..
</Stmt>
```

[1] Moderate blanks and new lines are inserted in the examples presented hereafter so that the readers easily see them.

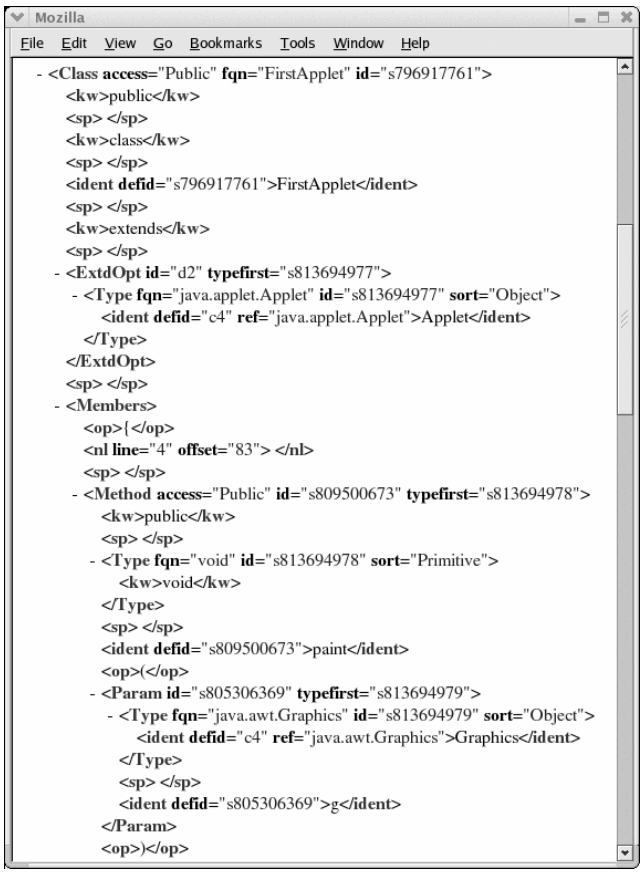

Figure 4. Tree view of the XSDML document.

The statement (`Stmt`) corresponding to the method call is decomposed into some detailed elements. Another example for the method declaration from line 5 to 7 in the source code is as follows:

```
<Method access="Public" id="s809500673"
        typefirst="s813694978">
  <kw>public</kw><sp> </sp>..
</Method>
```

The value of the `access` attribute and the textual content in the `kw` element are redundant. As shown in the above

two examples, the XSDML representation does not require lexical analysis any longer although it retains the contents of the original code. Moreover, these examples indicate that our conversion is suitable for implementing tools which manipulate actual source code and browse it without changing its appearance since white spaces (tabs and blanks) and new lines remain. For example, a refactoring tool or a coding checker prefers to use our representation. The highly abstract representation such as the JavaML is insufficient to implement these software tools although it is convenient for making a survey of source code and some tools require such representation independent to a specific programming language.

2.2. Semantic Analyzer

The significant feature of the Sapid/XML is that it reflects information based on semantic analysis in its XML representation. The semantic analyzer inserts two kinds of the information: type and reference. The type information is expressed by the `fqn` attribute. For the type Graphics at line 5 in the source code shown in Figure 2, the following description is generated.

```
<Type fqn="java.awt.Graphics" id="s813694979"
      sort="Object">
  <ident defid="c4" ref="java.awt.Graphics">
    Graphics</ident>
</Type>
```

It can be easily seen that the fully-qualified name of Graphics is java.awt.Graphics because of the value of `fqn`. The fully-qualified name is determined based on the search path for Java class and jar files, and used for obtaining the next reference information.

The reference information is classified as a local or global link. The local link is expressed by both the `id` and `defid` attributes like the JavaML. The `defid` indicates the link of the call or access to the element the `id` value of which equals to the `defid` value. A referenced element is always decided since the value of `id` must be unique within an XML document. The XSDML enhances this notation to express global links across several XML documents by adding the `ref` attribute. For example, the following description:

```
<Expr id="s830472195" sort="MethodCall">
  <ident defid="c302" fqn="void"
         ref="java.awt.Graphics">
    drawString</ident>
  <op>(</op>..
</Expr>
```

indicates invocation to the method drawString the `id` value of which equals to `c302` in the class java.awt.Graphics. The `fqn` attribute denotes the return type. The link of the field access is represented in the same manner.

Along with the reference information, `read` and `write` attributes are added to the `Expr` elements corresponding to all references to fields and local variables. For example, a reference to a variable g which is a prefix to the method call to drawString is represented as follows:

```
<Expr id="s830472194" sort="VarRef"
      read="yes" write="yes">
  <ident defid="s805306369">g</ident>
</Expr>
```

The `read="yes"` or `write="yes"` means that a variable is used without or with (possibly) changing its value, respectively.

The process of determining which method would be called and which field would be accessed is similar to that done when compiling source code. It is based on the apparent (or declarative) type of a related object since an actual object is decided at run-time and its precise type is not known at compile-time. The apparent type is obtained from the value of the `fqn` attribute corresponding to a primary `ident` or `Expr` element. Here the careful readers will wonder why java.awt.Graphics has the `id` attribute. Sapid/XML uses the byte code engineering library (BCEL) [9] and automatically generates summary XML documents from Java class and jar files whenever the files are referred by the analyzed class. Moreover, it determines which classes should be re-analyzed when a specified class is changed, by utilizing the global link information (and adding the new tags `Ances` and `FqnMap`). If any ancestor of the specified class, any class it refers to, or itself is modified, the platform automatically re-generates a new XSDML document from it.

The type and reference information (plus the read/write information) is often extracted by existing tools but is not reusable in general. For example, most compilers lose part of it after generating final class files. Although some of them store it in the class files, its format is hard to read because of optimization. Sapid/XML makes such information more explicit and provides it in an easy-to-use format in order that software tools can query and manipulate source code. This is significant since a semantic analyzer is hard and expensive to build from scratch. Moreover, the provided link information must be common and fundamental to all kinds of software tools although it is not enough to build them without supplemental information. Neither JavaML nor OOML contains the link information while JavaML deals with only local references.

2.3. Access Libraries and Wrappers

Every XSDML document is stored in the Java-XML repository. Tools running on Sapid/XML can request the access libraries to convert Java programs into XSDML documents and to retrieve some of them from the repository

with several queries. The retrieved documents can be used through various XML utilities, (e.g., the document object model (DOM) [1], the simple API for XML (SAX) [8], the extensible stylesheet language (XSL) and XSL transformations (XSLT) [10], and JDOM [4]). For example, the following Java code using DOM APIs outputs the name of all methods existing in a Java source file of interest.

```
Element elem = doc.getDocumentElement();
NodeList nl =
  elem.getElementByTagName("Method");
for (int i = 0; i < nl.getLength(); i++) {
  NodeList nl2 = nl.item(i).getChildNodes();
  for (int j = 0; j < nl2.getLength(); j++) {
    Node node = nl2.item(j);
    if (node.getNodeName().equals("ident")) {
       System.out.println(
         node.getFirstChild().getNodeValue());
    }
  }
}
```

The `doc` variable indicates a document object of the XSDML document generated from the source file.

The standard APIs (e.g., DOM and SAX) are of course convenient for writing code independent to a specific programming language but too primitive for most developers when they build a tool in practice. Accordingly, the developers tend to write tedious code repeatedly. To avoid this repetition, Sapid/XML provides several Java wrappers which have alternative, high-level APIs for accessing XSDML documents. In Figure 5, the rectangles denote the Java wrappers corresponding to XSDML elements depicted in the top of them.

The wrappers are classes tool developers would frequently use and allow them to easily access to portions of a DOM tree in the Java object form. For example, the code getting a list of classes in the Java source file (indicated by `doc`) is as follows:

```
Element elem = doc.getDocumentElement();
JavaFile jfile = new JavaFile(elem);
JavaClassList clist = jfile.getAllClasses();
```

In addition, the code outputting the name of all methods existing in a class is as follows:

```
JavaMethodList mlist = jclass.getAllMethods();
Iterator it = mlist.iterator();
while (it.hasNext()) {
  JavaMethod jm = (JavaMethod)it.next();
  System.out.println(jm.getName());
}
```

The variable `jclass` is an object of the JavaClass wrapper. All wrappers are designed only to extract information from XSDML documents and never change their contents. They are also useful samples of writing code that accesses and manipulates XSDML documents.

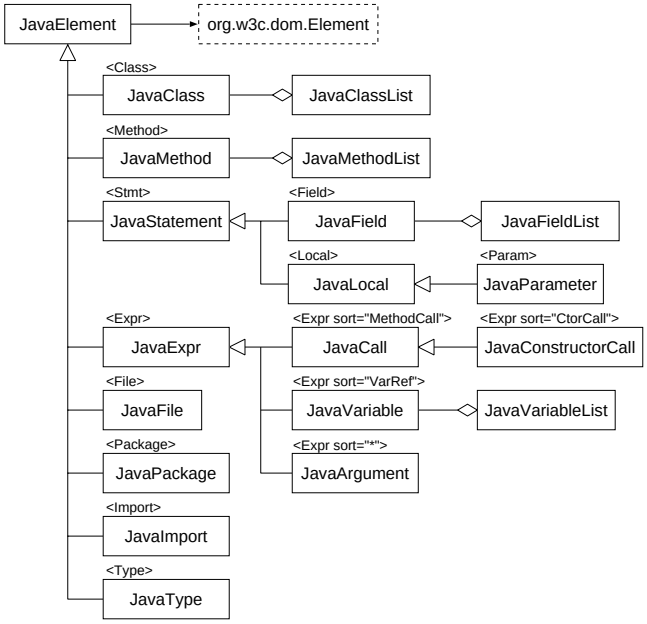

Figure 5. Java wrappers for the XSDML.

3. Practical Tools Using Sapid/XML

One strength of Sapid/XML is that it structures Java source code with several tags and embeds additional information resulting from semantic analysis in the converted XML documents. By specifying tags in querying and transformation, the portion of code can be accessed and extracted. Moreover, Sapid/XML neither loses tokens of original source code nor adds superfluous texts to the textual contents of elements when generating XSDML documents. This feature is convenient for modifying only the part of source code and retaining the remaining code, or marking (or highlighting) source code without changing its appearance. Most source code viewers and editors do not desire a tool platform to arbitrarily change the contents of source code (e.g., indentations or the position of braces) since they have their individual formatters.

To evaluate these benefits, we have developed the following tools.

- A method viewer generating a HTML document that lists the declaration of methods for each class.
- A source code browser generating a browsable code containing hyperlinked references in HTML form.
- A CFG/PDG constructor producing a control flow graph (CFG) [13] and a program dependence graph (PDG) [16] for each method existing in source code.
- A cross-reference extractor collecting link information about inverse references (e.g., callers of a method) and

Figure 6. Viewing the declaration of methods.

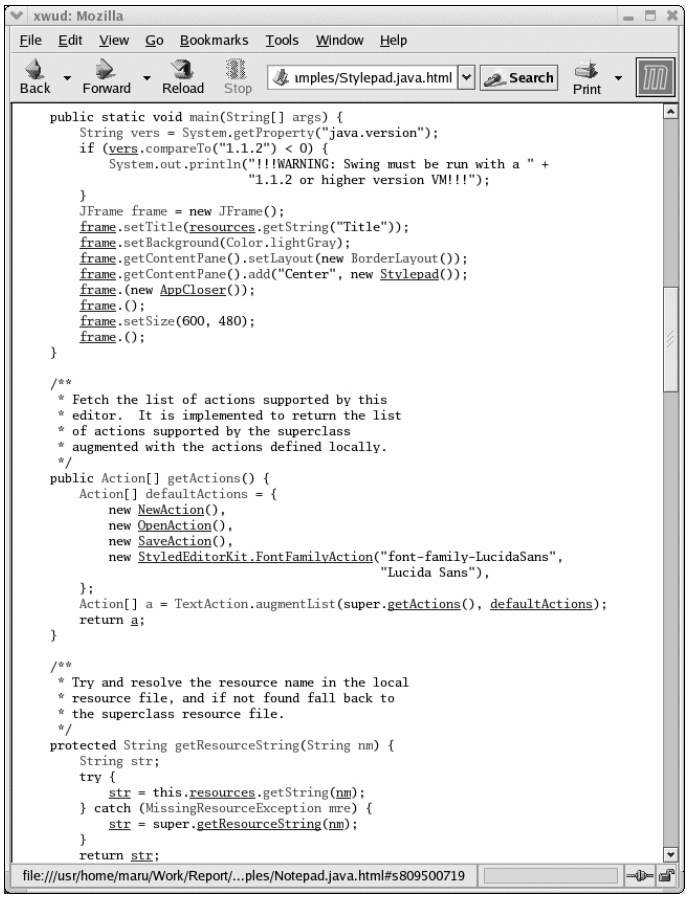

Figure 7. Viewing HTML-based source code.

relationships (e.g., method override), and producing XML documents containing the information.

- A refactoring browser restructuring existing source code without changing its observable behavior.

Due to space limitation, we will explain only the former three tools in this paper. [2]

3.1. Method Viewer

The method viewer is a simple XSLT application. Figure 6 shows a web browser displaying method declarations in the source code. It was trivial to identify classes, methods, and constructors since they were marked with `Class`, `Method`, and `Ctor` in the converted XSDML document, respectively. Carefully looking at Figure 6, all class names (type names) in the method declarations were replaced with fully-qualified ones. Displaying such information is easily performed by using the value of the `fqn` attribute of `Type` elements instead of its textual contents. With Sapid/XML, tools can obtain various kinds of information about source code through XML utilities and thus developers can build such tools without writing much code.

[2] All of these tools can be downloaded from http://www.jtool.org.

3.2. Source Code Browser

The source code browser is also an XSLT application. The developed stylesheet is described in Appendix A. Figure 7 shows a view of the generated HTML-based source code. This stylesheet performs mainly two transformations. One is to enclose the name (`ident`) of classes, methods, fields, local variables with the `<a name="{@defid}">` and `</a>` elements. The `@defid` indicates the value of the `defid` attribute of elements owning the enclosed names.

The other transformation is to find references to classes (types), methods, fields, and local variables, and enclose the references with `<a href="{$relpath}{$path}.html#{@defid}">` and `</a>` elements. As mentioned in Section 2.2, all references in XSDML documents have the `defid` attribute the value of which indicates the target element, which substitutes for `@defid`. Moreover, global references (other than references to local variables) have the `ref` attribute which denotes the fully-qualified name of a class containing the target element. The `$path` is obtained through the `FqnMap` map storing the correspondences between the fully-qualified name of

a class and the name of a file containing the class. The `$relpath` denotes a relative path to the top of directories storing HTML files and is provided as a parameter of the stylesheet.

Tags except for newly added ones are removed and the textual contents of all elements are left behind. A remarkable point is that the appearance of the restored source code is the exactly same as that of the original source code. Sapid/XML is well suited for creating this kind of tool because it preserves all tokens of the original source code in converted XSDML documents.

3.3. CFG/PDG Constructor

The CFG and PDG (or control and data flow) are often used for creating tools that support software development. For example, the CFG is useful for eliminating dead code or code clone, and the PDG is invaluable for debugging or testing. Program slicing [20] is famous application using the PDG, which is widely applied to various fields. Incidentally, our developed refactoring browser uses the CFG and PDG. The information about CFGs and PDGs can be obtained through XML documents and/or Java objects.

The CFG consists of a set of nodes and edges. Each node denotes a statement which is either an assignment or a condition predicate, which is marked the `Stmt` or `Expr` tag. Each edge represents immediately control flow from a statement and another one. An example of the generated CFG is as follows:

```
<nodes>..
  <node no="4" id="s805306373">
    <def-var id="s805306373" name="sum"/></node>
  <node no="5" id="s826277895">
    <use-var id="s805306372" name="n"/></node>
  ..</nodes>
<edges>..
  <edge src="5" dst="6" sort="TrueCtrlFlow"/>
  <edge src="6" dst="7" sort="TrueCtrlFlow"/>
  <edge src="7" dst="5" sort="TrueCtrlFlow"
        loopback="yes"/>
  <edge src="5" dst="8" sort="FalseCtrlFlow"/>
..</edges>
```

The `src` or `dst` attribute demotes the value of the `no` attribute of a source or destination node, respectively. The `sort` attribute is `TrueCtrlFlow` (if-then), `FalseCtrlFlow` (if-else), or `FallThrFlow` [15]. The `loopback="yes"` means its edge is a back-edge for a loop. The analyzer of the current version of Sapid/XML cannot deal with control flow involved in exception. To alleviate this problem, a path edge [17] which indicates control flow for exception handling will be embedded.

Similar to the CFG, the PDG consists of a set of nodes and edges. Each node is equal to a node of the CFG generated from the same source code. Edges denote control and data dependences. A control dependence edge represents a control condition on which the execution of a statement depends. Data dependence edge represents flow of data between statements, which is classified as either loop-carried or loop-independent [18]. An example of the generated PDG is as follows:

```
<nodes>..</nodes>
<edges>
  <edge src="5" dst="6" sort="TrueCtrlDep"/>
  <edge src="5" dst="7" sort="TrueCtrlDep"/>
  ..
  <edge src="2" dst="5" sort="ParameterIn">
    <var id="s805306372" name="n"/></edge>
  <edge src="4" dst="6" sort="DefUseDep">
    <var id="s805306373" name="sum"/></edge>
  <edge src="6" dst="6" sort="DefUseDep">
    <var id="s805306373" name="sum" lc="5"/></edge>
  <edge src="8" dst="9" sort="ParameterOut">
    <var id="s809500674" name="$sum"/></edge>
</edges>
```

The `sort` attribute equals `TrueCtrlDep` (true control dependence), `FalseCtrlFlow` (false control dependence), `DefUseDep` (def-use data dependence), `ParameterIn` (def-use data dependence related to an incoming parameter), or `ParameterOut` (def-use data dependence related to an outgoing parameter). The `lc` attribute in a `var` element indicates a loop-node carrying the edge enclosing the `var` element.

3.4. Discussion

Both of the former two tools were completed with little time and effort and comprised small amount of description (about 46 LOC and 55 LOC, respectively) because we were able to use an existing XSL processor and wrote code in the standardized stylesheet language without learning proprietary programming interfaces. The CFG/PDG constructor demonstrated that Sapid/XML provides sufficient information about source code, which is not inferior to that provided by the AST. Moreover, we conformed that extending the original XSDML representation is useful for sharing and exchanging analyzed information. In fact, one new attribute to indicate a location of each code fragment was added when developing the cross reference extractor. Moreover, one new tag and one new attribute to express the changes of source code were defined in the refactoring browser.

In addition to these tools, we are considering tools that enable developers to annotate any code fragment by using individual elements or attributes. For example, a version control tool might desire to attach an annotation containing information about the last modified time to not only each file but also each method as follows:

```
<Method .. modified="Mon Apr 5 10:45:14">..
</Method>
```

or it might present access permission for each method as follows:

```
<Method .. mode="Read-only">..
</Method>
```

Table 3. Size of converted XML documents and processing time.

Program	Java source file (.java)			XML file (.xml)			
	# of files	LOC	Size [bytes]	.java.xml [bytes]	ratio	.class.xml [bytes]	Total [bytes]
Notepad	2	1,343	38,805	412,522	10.63	1,239,167	1,651,689
Sylepad	5	2,159	65,245	717,249	11.00	1,433,866	2,151,115
SwingSet2	31	8,617	294,619	3,088,867	10.48	2,193,784	5,282,651
Java2D	62	14,187	509,949	6,355,034	12.46	2,217,521	8,572,555

Table 4. Size of converted XML documents and processing time.

Program	XML file (.xml)		Conversion time [s]				Manipulation time [s]			
	# of files	# of elements	Syntactic	Semantic	Total	Each file	Counter	Each file	Viewer	Each file
Notepad	2	20,634	5.444	22.077	27.521	13.761	0.032	0.016	2.250	1.125
Sylepad	5	35,418	9.336	28.793	38.129	7.626	0.034	0.007	4.970	0.994
SwingSet2	31	150,086	59.455	140.051	199.506	6.436	0.068	0.002	27.220	0.878
Java2D	62	308,631	88.526	480.403	568.929	9.176	0.129	0.002	54.940	0.886

Additionally, developers might want to embed a temporary note that differs from a conventional (permanent) comment into source code.

```
<Method note="n000000001">..</Method>
----
<note id="n000000001" expire="04/12/2004">
The name of this method was recently changed.
</note>
```

In this case, the unparser must be slightly modified and a proper editor (or viewer) displaying the textual contents of the added `note` tag is needed to prepare.

4. Experimental Results

The XML representation of source code in general causes expansion of the file size and processing time because of its portability and flexibility. To roughly evaluate performance of Sapid/XML, we carried out simple experiments with four programs (Notepad, Sylepad, SwingSet2, and Java2D) packaged in the Sun Microsystems J2SDK1.4.2.

Table 3 shows the size of the original Java source files and their converted XSDML files. The size of the converted XML files (**.java.xml**) is about 10 times (10.63, 11.00, 10.48, and 12.46 times, respectively) larger than that of original files. This is because our proposed XML representation contains various kinds of analyzed information of source code. Moreover, Sapid/XML automatically generates summary XSDML documents (**.class.xml**) from classes related to the analyzed Java source files. These files consume much space although they can be shared by respective programs. The repository size might cause the "out of memory" problem when Sapid/XML handles a huge program.

Table 4 shows two types of the processing time. The conversion time denotes how long it takes to convert Java source files into XSDML documents. This time is divided into two phases: syntactic parsing and semantic analysis. The manipulation time were measured by using two applications. The "Counter" application traverses all elements (tags and attributes) and counts their numbers, which uses the DOM processor, Xerces2 Java Parser 2.6.2 [12]. The "Viewer" application generates a browsable source code in HTML form. It uses the XSL processor, Xalan Java version 2.6.0 [11] and the stylesheet described in Appendix A. The execution was performed on a computer with a Pentium4 2.4GHz CPU and a 640MB of RAM, running Red Hat Linux9 and Sun Microsystems J2RE1.4.2_01.

The conversion time for each Java source file is about 6 to 14 seconds and is much longer than the general compile time. This main reason is that Sapid/XML uses XSDML documents and an XML processor when performing global semantic analysis. This result might not be serious to build an application which seldom needs the conversion (e.g., a source code viewer for the standard libraries) but is much considerable to build interactive tools which need the frequent re-conversion. To reduce the conversion time, we are planning to adopt the semantic analyzer of sophisticated compilers or modifying existing IDEs to generate XSDML documents. The manipulation time for simple applications (not complicated applications such as the semantic analyzer) is considered to be reasonable (about 1 second for each file).

5. Conclusion

Tool developers require more portable and extensible representations of tool platforms. This paper has proposed the XSDML representation using XML and Sapid/XML

that is a tool platform for managing such representation. Sapid/XML retains original code fragments in the converted XSDML documents and inserts the globally analyzed information into them. With this platform, the developers easily build software tools that collaborate with each other.

For the platform to be truly practical, its performance must be improved and the development of many tools are needed. From functional points of view, Sapid/XML cannot replace an existing powerful IDE. Additionally, our proposed XSDML representation is not perfect and should be refined. We are planning to integrate the XSDML representation and its converter into popular IDEs (e.g., Eclipse [2]).

The Sapid/XML tool platform and some tools running on it can be downloaded from http://www.jtool.org.

Acknowledgments

The authors would like to thank Akinori Yonezawa at the University of Tokyo and Etsuya Shibayama at the Tokyo Institute of Technology for their helpful suggestions and comments. This work is based on and has been cooperating with the Sapid project. We also thank Kiyoshi Agusa of the Nagoya University and all members who have been engaging the Sapid project. This work was sponsored by the Information-technology Promotion Agency (IPA), Japan.

References

[1] Document Object Model (DOM). `http://www.w3.org/DOM/`.

[2] Eclipse. `http://www.eclipse.org/`.

[3] Extensible Markup Language (XML). `http://www.w3.org/XML/`.

[4] JDOM. `http://www.jdom.org/`.

[5] Mizilla. `http://www.mozilla.org/`.

[6] RECORDER. `http://recoder.sourceforge.net/`.

[7] Semantic Designs, Inc., The DMS Software Reengineering Toolkit. `http://www.semdesigns.com/Products/DMS/DMSToolkit.html`.

[8] Simple API for XML (SAX). `http://www.saxproject.org/`.

[9] The Byte Code Engineering Library (BCEL). `http://jakarta.apache.org/bcel/`.

[10] The Extensible Stylesheet Language Family (XSL). `http://www.w3.org/Style/XSL/`.

[11] Xalan-Java. `http://xml.apache.org/xalan-j/`.

[12] Xerces2 Java Parser. `http://xml.apache.org/xerces2-j/`.

[13] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[14] G. J. Badros. JavaML: A markup language for java source code. In *Proc. Int'l WWW Conference*, May 2000. `http://www9.org/w9cdrom/index.html`.

[15] T. Ball and S. B. Horwitz. Slicing programs with arbitrary control flow. In *Proc. Intl. Work. on Automated and Algorithmic Debugging*, LNCS 749, pages 206–222, May 1993.

[16] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimizati- on. *ACM TOPLAS*, 9(3):319–349, July 1987.

[17] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proc. OOPSLA*, pages 312–326, Oct. 2001.

[18] S. Horwitz, T. Ball, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–60, Jan. 1990.

[19] E. Mamas and K. Kontogiannis. Towards portable source code representations using xml. In *Proc. Working Conference on Reverse Engineering*, pages 172–182, Nov. 2000.

[20] M. Weiser. Program slicing. *IEEE Trans. Software Engineering (TSE)*, 10(4):352–357, July 1984.

A. XSLT Stylesheet Used in the Experiments

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:param name="relpath"/>
<xsl:key name="Fqn" match="FqnMap" use="@fqn"/>

  <xsl:template match="/">
    <html><pre><xsl:apply-templates/></pre></html>
  </xsl:template>
  <xsl:template match="*|@*">
    <xsl:apply-templates select="*|@*|text()"/>
  </xsl:template>
  <xsl:template match="text()">
    <xsl:value-of select="."/>
  </xsl:template>

  <xsl:template match="Class/ident|Intf/ident|
                       Method/ident|Ctor/ident|
                       Field/Expr/ident|
                       Local/Expr/ident|
                       Param/ident" priority="1">
    <a name="{@defid}">
      <font color="red">
        <xsl:value-of select="."/></font></a>
  </xsl:template>

  <xsl:template match="Type[@sort='Object']/ident|
                       Expr[@sort='VarRef']/ident|
                       Expr[@sort='MethodCall']/ident|
                       Expr[@sort='CtorCall']/ident">
    <xsl:choose>
    <xsl:when test="@ref">
      <xsl:variable name="path"
                    select="key('Fqn',@ref)/@path"/>

      <xsl:if test="contains($path, '.java')">
        <a href="{$relpath}{$path}.html#{@defid}">
          <xsl:value-of select="."/></a>
      </xsl:if>

      <xsl:if test="contains($path, '.class')">
        <font color="green">
          <xsl:value-of select="."/>
        </font>
      </xsl:if>
    </xsl:when>

    <xsl:otherwise>
      <a href="#{@defid}">
        <xsl:value-of select="."/></a>
    </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```