IEICE TRANS. INF. & SYST., VOL.E89–D, NO.7 JULY 2006

PAPER

A Tool Platform Using an XML Representation of Source Code Information

Katsuhisa MARUYAMA[†a)] ***and*** **Shinichiro YAMAMOTO**[††]**,** ***Members***

SUMMARY Recent IDEs have become more extensible tool platforms but do not concern themselves with how other tools running on them collaborate with each other. They compel developers to use proprietary representations or the classical abstract syntax tree (AST) to build source code tools. 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 source code, since the original code is annotated with tags based on its structure and retained within the tags. 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 the source code, and also enables them to build new tools by using existing XML utilities.

key words: *source code representation, tool platform, Java, XML, program analysis, source code manipulation tools*

1. Introduction

Object-oriented software is hard to develop without integrated development environments (IDEs) since it consists of many classes and contains various kinds of relationship between them. A significant point is that a recently released IDE is not only a collection of programming tools but also an extensible tool platform. For example, Eclipse [1] has a powerful plug-in mechanism for easily adding new tools to itself and removing existing tools from itself.

By supporting the plug-in mechanism, developers (and researchers) have a chance to build their own tools and would want 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 developers' diverse requirements. Unfortunately, conventional tool platforms store information about source code by using either proprietary representations or the typical abstract syntax tree (AST) [2]. Of course these representations contain sufficient information and several powerful tool platforms such as Eclipse [1], the DMS Software Reengineering Toolkit [3], or RECODER [4] provide well-designed application programming interfaces (APIs) for accessing the information.

However, the classical representations are neither portable nor extensible. That is, most of the conventional platforms do not concern themselves with how a newly built tool stores the additional information obtained through its execution and exchanges such information with other tools. In addition, proprietary APIs are insufficient for building diverse tools. The tool developers tend to create overhead modules which are used for extracting necessary information from the integrated representation in their respective tools, or they might have to modify the integrated modules and date structure. To build various kinds of software tools managing source code and make them collaboratively work together, the tool platform should not only use a simple standard but also portable and extensible representation. Such a representation would act as a medium for exchanging source-code information and would allow 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 extensible markup language (XML) [5]. This paper proposes the tool platform, which is called Sapid/XML (sophisticated APIs for CASE tool development with an XML repository) and the extension of XSDML (extensible software document markup language) [6][*]. Sapid/XML converts source code into an XSDML document using fine-grained XML representation. Code fragments are classified by marking them with respective tags and are 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.

Accordingly, Sapid/XML makes the source code more portable and convenient since XSDML exposes the structure and relationship found in the source code and the format is based on XML. XML is a simple, widely used text-based format that is used to design markup languages suitable for the capture and exchange of information. Many existing XML utilities can be used for examining and manipulating the source code. Sapid/XML also allows developers to extend the prepared representation although its extension would need a simple consistency check for the document type definition (DTD) [5]. They can define new tags and attributes to share common information and exchange specific information. It is useful for building new software tools to extend the prepared representation of source code with-

Manuscript received July 19, 2005.

[†]The author is with the Department of Computer Science, Ritsumeikan University, Kusatsu-shi, 525–8577 Japan.

[††]The author is with the Department of Information Systems, Aichi Prefectural University, Aichi-ken, 480–1198 Japan.

a) E-mail: maru@cs.ritsumei.ac.jp

DOI: 10.1093/ietisy/e89–d.7.2214

[*]The extended version is accurately defined as XSDML level3 but we simply call it XSDML in this paper.

out examining and modifying modules in the tool platform. Here the authors have to mention that Sapid/XML does not intend to replace existing IDEs. It shows the potential of a tool platform using an XML representation which is an alternative to the classical AST of source code.

We first describe existing XML representations of object-oriented source code. Next we 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 document. Then we show several software tools running on Sapid/XML and give experimental results in respect to the performance of Sapid/XML. Finally, we conclude with a summary.

2. XML Representations of Source Code

Several XML formats currently exist for representing object-oriented source code. For example, GraX [7] and GXL [8] are classified into a graph-based format. They store information about nodes and edges of the abstract syntax graph (ASG) [9] without reflecting the nested structure of the source code in its XML representation. On the other hand, Harmonia [10], JavaML [11], OOML (cppML and JavaML) [12], bison-based parser [13], XMLizer [14], FreeTXL [15], and srcML [16], [17] directly encode actual source code or its AST to the nested structure of their XML representations.

These representations are all intended to exchange information about source code or display the abstract structure of the source code. They are fulfilling such purpose since XML is a simple, extensible, widely used text-based format. This concept is analogous to that of our used XSDML. However, the main purpose of Sapid/XML is to facilitate developers in building tools for supporting software development.

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 browser or a code checker works well with our representation since most tool users do not want it to remove comments or formatting characters (white spaces and new lines) from the original source code. The highly abstract representation such as JavaML is insufficient to implement these software tools although its representation would be convenient for making a survey of the source code or measuring its metrics, and some tools requiring such a representation independent to a specific programming language.

From this point of view, XSDML is closely related to srcML although their target programming languages are different (Java and C++, respectively). All of the conventional representations except srcML and XSDML never support the representation of comments or formatting [16]. That is, only srcML and XSDML preserve the original text of source code containing formatting information and guarantee the restoration of the complete original source code. Moreover, both of them have several common features: tag names based on programmers' knowledge (i.e., syntactic names such as classes, methods, or fields) and the conversion that directly inserts tags to source code as meta-data.

While XSDML is similar to srcML, there are three main differences between them. First, XSDML provides all fragments of source text (operators or separators, identifiers, keywords, white spaces, and new lines) with dedicated tags (see Sect. 3.1). These tags allow developers or tools to add extra white spaces and new lines that were not contained in the original source code. The original white spaces and new lines are always enclosed with terminal elements while extra ones are enclosed with non-terminal elements. Secondly, XSDML aggressively exploits many kinds of attributes while very few attributes are used in srcML (see Sect. 3.1). The verbose attributes alleviate additional lexical analysis of the contents of elements or the time-consuming traversal of several elements when the developers and tools obtain the properties of code fragments (e.g., modifiers). Finally, XSDML contains several useful links obtained through global (and local) semantic analysis for the whole of source code (Sect. 3.2). Some of the links are used in GXL or JavaML but are not provided by srcML.

3. Sapid/XML Tool Platform

Sapid/XML generates XML documents represented in our proposed XSDML from Java programs (written in Java 1.4 or earlier) 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 these documents. The section also describes the access libraries and the Java wrappers accessing the documents.

3.1 Syntactic Parser

The original XSDML was proposed in [6]. XSDML represents the classical text-based source code as 20 non-terminal elements and 7 terminal ones. The terminal element has only the textual contents while the non-terminal element can nest others. The syntactic parser directly inserts 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 the 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 the tags and leaving behind the textual contents of the terminal elements. The attributes are used for expressing additional properties such as modifiers, accessibility settings, fully-qualified names, and sorts of elements. For example, the type (`Type`), statement (`Stmt`), expression (`Expr`), and literal (`literal`) elements are clas-

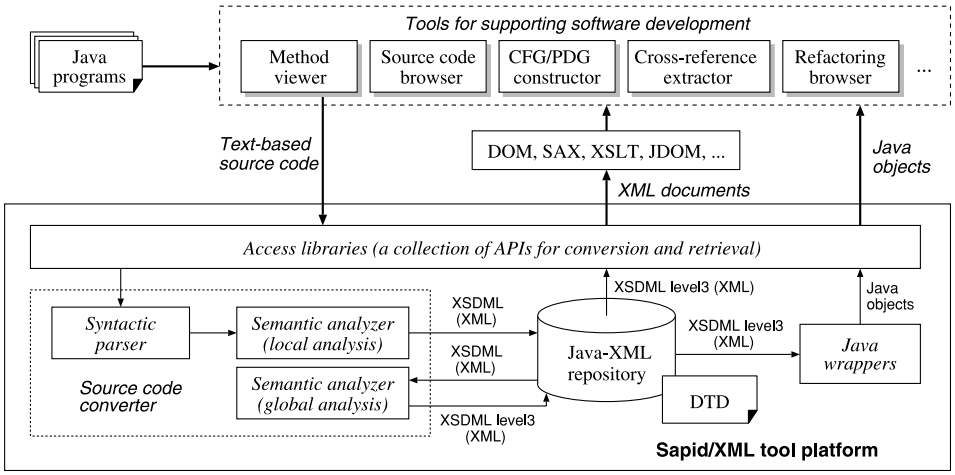


Fig. 1 Overview of the proposed tool platform.

```
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: }
```

Fig. 2 Sample Java source code.

sified as 3, 15, 59, and 6 by the attribute `sort`, respectively [†].

The simple source code quoted from [11] and a tree view of XSDML document converted from it are shown in Figs. 2 and 3[††], respectively. The original code can be seen in the textual contents of the terminal elements (e.g., the blanks or keywords are enclosed with the <sp> or <kw>).

3.2 Semantic Analyzer

The noteworthy feature of Sapid/XML is that it reflects information based on semantic analysis in its XML representation. The semantic analyzer inserts two kinds of 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 Fig. 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 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

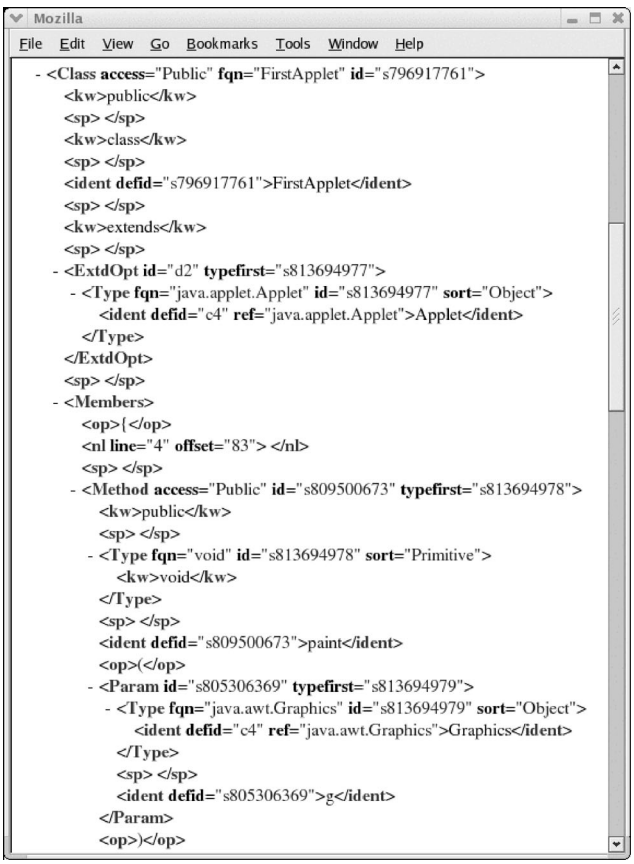


Fig. 3 Tree view of the XSDML document.

`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 `id` value must be unique within an XSDML document. XSDML enhances this notation to express

[†]See `http://www.jtool.org/xsdml.html` for details.

[††]The crude document of XSDML is hard for a human to read but we can use various XML utilities to view it. The figure is displayed by the Mozilla [18].

global links across several XML documents by adding the `ref` attribute. For example, the following description:

```
<Expr id="s830472195" sort="MethodCall">
  <ident defid="c302" ref="java.awt.Graphics"
         fqn="void">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, the `read` and/or `write` attributes are added to all `Expr` elements corresponding to the references to fields and local variables. For example, the variable g which is a primary expression of 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 the value of the variable is used or changed, respectively. The `write="yes"` is also added when the state of the object indicated by the reference variable might change (i.e., the value of any fields defined in the object might be modified).

The process of determining which method would be called and which field would be accessed is similar to that done when compiling source code (Sects. 15.11 and 15.12 in [19]). 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 the 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) [20] and automatically generates summary XML documents from class (and jar) files whenever the files are referred by the analyzed class. Moreover, it determines which classes should be re-analyzed when a class is changed, by utilizing the global link information (and specially adding the new tags `Ances` and `FqnMap`). If any ancestor of the specified class, any class it refers to, or itself is modified, a new XSDML document is automatically regenerated from it.

The type and reference (plus read/write) information is often extracted by existing tools but is not reusable in general. For example, most compilers lose part of the information after generating final class files. Although some of them store the information 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 easily query and manipulate source code. This is significant since such semantic analyzer is 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.

3.3 Access Libraries and Wrappers

Every XSDML document is stored in the Java-XML repository. Tools running on Sapid/XML can request access to the libraries to convert Java programs into XSDML documents or 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) [21], the simple API for XML (SAX) [22], the extensible stylesheet language (XSL) and XSL transformations (XSLT) [23], and JDOM [24]). 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 tools in practice. Accordingly, the developers tend to write tedious code repeatedly. To avoid this repetition, Sapid/XML provides several Java wrappers which have high-level APIs for accessing XSDML documents. In Fig. 4, the rectangles denote the wrappers corresponding to the XSDML elements depicted in the top of them.

The wrappers are classes that tool developers would frequently use and allow them to easily access 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 the XSDML documents.

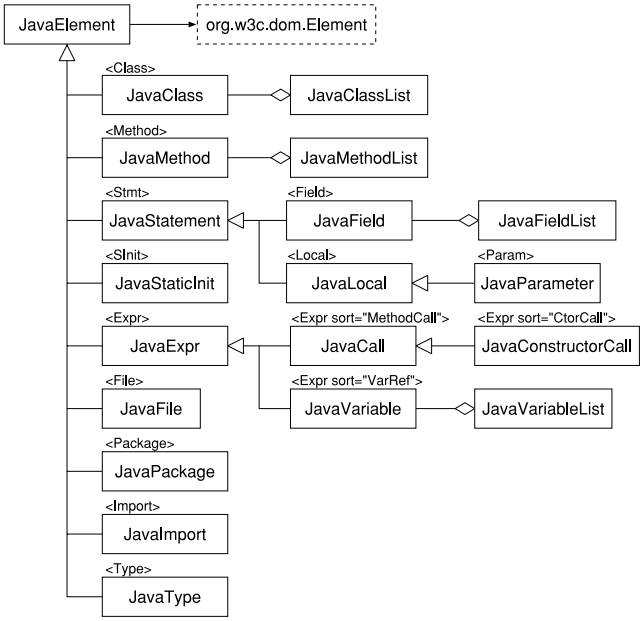


Fig. 4 Java wrappers for XSDML.

4. 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, portions of the 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 terminal 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 tool platforms 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 listing 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) [2] and a program dependence graph (PDG) [25] for each method existing in source code.
- A cross-reference extractor collecting link information about inverse references (e.g., callers of a method) and relationships (e.g., method override), and producing XML documents containing the information.
- A refactoring browser [26] restructuring existing source code without changing its observable behavior.

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

Fig. 5 Viewing the declaration of methods.

4.1 Method Viewer

The method viewer is a simple XSLT application. Figure 5 shows a web browser displaying method declarations in 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 Fig. 5, all class (or 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 their actual names. 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.

4.2 Source Code Browser

The source code browser is also an XSLT application. The stylesheet is described in Appendix. Figure 6 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 sandwiched names.

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

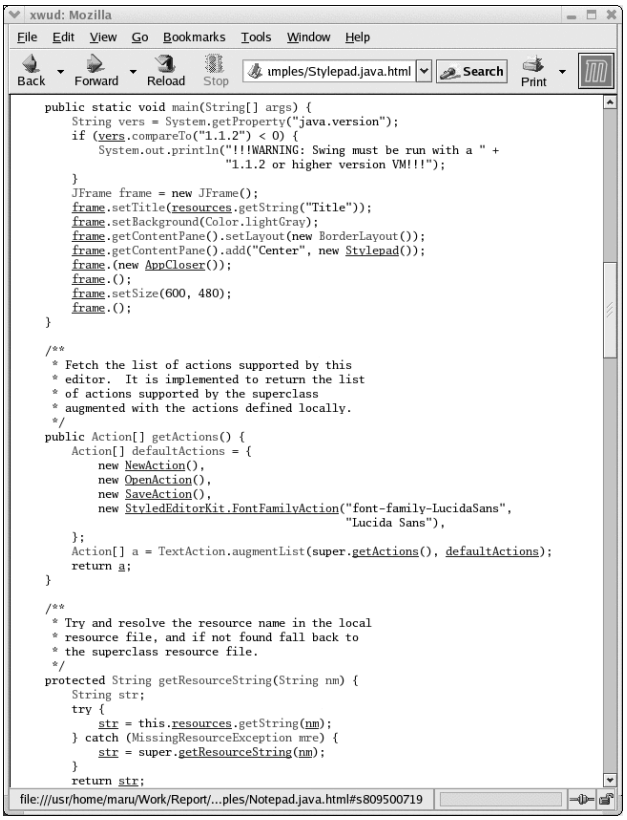


Fig. 6 Viewing HTML-based source code.

The other transformation is to find references to classes, methods, fields, and local variables, and enclose the references with and elements. As mentioned in Sect. 3.2, all references in XSDML documents have the `defid` attribute, the value of which indicates the target element and substitutes for `@defid`. Moreover, global references (other than references to local variables) have the `ref` attribute which indicates 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 significant 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.

4.3 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

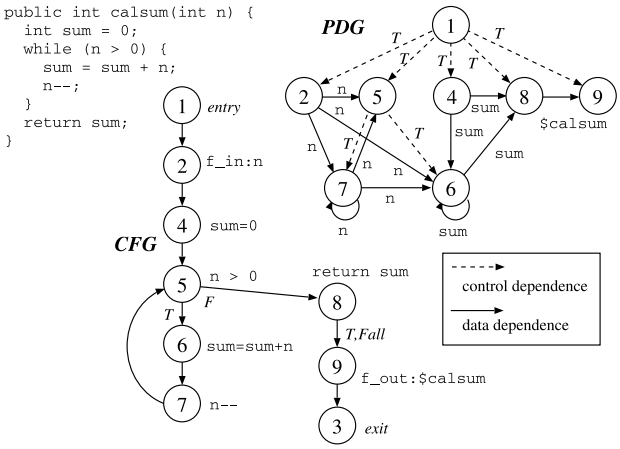


Fig. 7 Source code, and its CFG and PDG.

code clone, and the PDG is invaluable for debugging or testing. Program slicing [27] is a famous application using the PDG, which is widely applied to various fields. Our developed refactoring browser uses this CFG/PDG constructor. The information about CFGs and PDGs can be obtained through both XML documents and Java objects.

Figure 7 shows source code of a method written in Java, and its CFG and PDG. 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 immediate control flow from a statement and another one. The representation of the generated CFG is as follows:

```
<CFG class="Sum" method="calsum( int )">
  <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>
</CFG>
```

The `src` or `dst` attribute denotes the value of the `no` attribute of a source or destination node, respectively. The `sort` attribute is either `TrueCtrlFlow` (if-then), `FalseCtrlFlow` (if-else), or `FallThrFlow` [28]. 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 [29] 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 corresponds to a node of the CFG generated from the same source code. Edges denote control and data dependences. A control dependence edge repre-

sents 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 [30]. The representation of the generated PDG is as follows:

```
<PDG class="Sum" method="calsum( int )">
  <nodes>..</nodes>
  <edges>..
    <edge src="5" dst="6" 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="$calsum"/></edge>..
  </edges>
</PDG>
```

The `sort` attribute equals either `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.

In general, the dependency analysis of the whole program is too expensive. Consequently, the current CFG/PDG constructor does not further analyze the variable appearing in the primary expression of method invocation or field access. Such a variable is considered to be modified and thus the created PDGs are all conservative. For example, consider the following code:

```
int x = obj.getX();
int y = obj.getY();
```

The variable `obj` indicates an object of the class defining the methods `getX()` and `getY()`. In this case, the CFG/PDG constructor produces a data dependence from the first statement to the second statement since the value of the variable `obj` is both read and written in each statement. Precisely, this data dependence is dispensable only if the execution of the method `getX()` never changes the state of the object `obj` (e.g., the method does not change the value of every field but only returns its value).

4.4 Discussion

Each of the former two tools was completed with little time and effort and comprised small amount of description (about 46 LOC and 52 LOC, respectively) because we were able to use an existing XSL processor and wrote code in a standardized and popular language without learning proprietary programming interfaces. Through development of the CFG/PDG constructor, we confirmed that Sapid/XML provides sufficient information about source code, which is not inferior to that provided by the AST. In addition, we confirmed that extending the original XSDML representation is useful for sharing and exchanging analyzed information. In fact, one new attribute was added in order to indicate locations of code fragments when we developed the cross reference extractor, and one new tag and one new attribute were used in the refactoring browser in order to express the changes of source code.

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

```
<Method modified="Mon Apr 4 21:11:57 JST 2005">..
</Method>
```

or it might assign an access permission for each method as follows:

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

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

```
<Method note="s809500673">..</Method>
..
<note defid="s809500673" expire="04/20/2005">
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 tag is needed to prepare.

5. 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, Stylepad, SwingSet2, and Java2D) packaged in the Sun Microsystems J2SDK1.4.2.

Table 1 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 about the source code. Moreover, Sapid/XML automatically generates summary XSDML documents (**.class.xml**) from classes related to Java source files. These files consume much space although they can be shared by respective programs. We do not think that the repository size is serious because recent computers have a large size of memory.

Table 2 shows two types of processing time. The conversion time denotes how long does it take to convert Java source files into XSDML documents. This time is divided into two phases: syntactic parsing and semantic analysis. The manipulation time was measured by using two applications. The "Counter" application traverses all elements

Table 1 Size of converted XSDML documents.

Program	Java source file (.java)			XSDML 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
Stylepad	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 2 Processing time for conversion and manipulation of XSDML documents.

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
Stylepad	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

(tags and attributes) and counts their numbers, which uses the DOM processor, Xerces2 Java Parser 2.6.2 [31]. The "Viewer" application generates a browsable source code in HTML form. It uses the XSL processor, Xalan Java version 2.6.0 [32] and the stylesheet described in Appendix. The execution was performed on a computer with a Pentium4 2.4 GHz CPU and a 640 MB 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 critical to build an application which seldom needs the conversion (e.g., a source code viewer or a software metrics tool) but it might be problematic to build interactive tools which need the frequent re-conversion. To reduce the conversion time, we are planning to adopt the semantic analyzer of a sophisticated compiler or modifying an existing IDE to generate XSDML documents.

6. Conclusion

Tool developers require more extensible and portable representations of source code. 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 a functional point 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 [1]). We have almost completed the integration of XSDML into Eclipse, and XML-based information about Java source code can be accessed through our provided Eclipse plug-in.

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

Acknowledgments

The authors would like to thank Akinori Yonezawa, Etsuya Shibayama, and all members who have been engaging the Sapid project including Kiyoshi Agusa. We also thank the members of the Institute for Software Research (ISR) at the University of California, Irvine (UCI), who give us helpful and valuable comments. Especially, we thank Christopher Van der Westhuizen and Ping H. Chen of UCI for their excellent suggestions that improve this paper. This work was partially sponsored by the Information-technology Promotion Agency (IPA), Japan.

References

[1] "Eclipse," http://www.eclipse.org/
[2] A.V. Aho, R. Sethi, and J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.
[3] Semantic Designs, Inc., "DMS software reengineering toolkit," http://www.semdesigns.com/Products/DMS/DMSToolkit.html
[4] "RECODER," http://recoder.sourceforge.net/
[5] "Extensible Markup Language (XML)," http://www.w3.org/XML/
[6] H. Yoshida, S. Yamamoto, and K. Agusa, "A generic fine-grained software repository using XML," IPSJ Journal, vol.44, no.6, pp.1509–1516, June 2003.
[7] J. Ebert, B. Kullbach, and A. Winter, "GraX — An interchange format for reengineering tools," Proc. Working Conf. Reverse Engineering (WCRE), pp.89–98, Oct. 1999.
[8] R.C. Holt, A. Winter, and A. Schürr, "GXL: Toward a standard exchange format," Proc. Working Conf. Reverse Engineering (WCRE), pp.162–171, Nov. 2000.
[9] "GXL: Graph eXchange Language," http://www.gupro.de/GXL/
[10] M. Boshernitsan and S.L. Graham, "Designing an XML-based exchange format for Harmonia," Proc. Working Conf. Reverse Engineering (WCRE), pp.287–289, Nov. 2000.
[11] G.J. Badros, "JavaML: A markup language for Java source code," Proc. Int'l WWW Conference, May 2000. http://www9.org/w9cdrom/index.html
[12] E. Mamas and K. Kontogiannis, "Towards portable source code representations using XML," Proc. Working Conf. Reverse Engineering (WCRE), pp.172–182, Nov. 2000.
[13] J.F. Power and B.A. Malloy, "Program annotation in XML: A parse-

tree based approach," Proc. Working Conf. Reverse Engineering (WCRE), pp.190–198, Oct. 2002.
[14] G. McArthur, J. Mylopoulos, and S.K.K. Ng, "An extensible tool for source code representation using XML," Proc. Working Conf. Reverse Engineering (WCRE), pp.199–208, Oct. 2002.
[15] J.R. Cordy, "Generalized selective XML markup of source code using agile parsing," Proc. Int'l Work. Program Comprehension (IWPC), pp.144–153, May 2003.
[16] J.I. Maletic, M.L. Collard, and A. Marcus, "Source code files as structured documents," Proc. Int'l Work. Program Comprehension (IWPC), pp.289–292, June 2002.
[17] J.I. Maletic, M. Collard, and H. Kagdi, "Leveraging XML technologies in developing program analysis tools," Proc. Adoption-Centric Software Engineering (ACSE), pp.80–85, May 2004.
[18] "Mozilla," http://www.mozilla.org/
[19] J. Gosling, B. Joy, and G. Steele, The Java Language Specification, Addison-Wesley, 1996.
[20] "Byte Code Engineering Library (BCEL)," http://jakarta.apache.org/bcel/
[21] "Document Object Model (DOM)," http://www.w3.org/DOM/
[22] "Simple API for XML (SAX)," http://www.saxproject.org/
[23] "Extensible Stylesheet Language Family (XSL)," http://www.w3.org/Style/XSL/
[24] "JDOM," http://www.jdom.org/
[25] J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The program dependence graph and its use in optimization," ACM Trans. Programming Language and Systems (TOPLAS), vol.9, no.3, pp.319–349, July 1987.
[26] K. Maruyama and S. Yamamoto, "Design and implementation of an extensible and modifiable refactoring tool," Proc. Int'l Work. Program Comprehension (IWPC), pp.195–204, May 2005.
[27] M. Weiser, "Program slicing," IEEE Trans. Softw. Eng. (TSE), vol.10, no.4, pp.352–357, July 1984.
[28] T. Ball and S.B. Horwitz, "Slicing programs with arbitrary control flow," Proc. Intl. Work. on Automated and Algorithmic Debugging, LNCS 749, pp.206–222, May 1993.
[29] 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," Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp.312–326, Oct. 2001.
[30] S. Horwitz, T. Ball, and D. Binkley, "Interprocedural slicing using dependence graphs," ACM Trans. Programming Language and Systems (TOPLAS), vol.12, no.1, pp.26–60, Jan. 1990.
[31] "Xerces2 Java Parser," http://xml.apache.org/xerces2-j/
[32] "Xalan-Java," http://xml.apache.org/xalan-j/

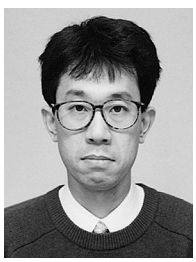
Appendix: XSL Stylesheets (htmlview.xsl)

```
<?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>
```



Katsuhisa Maruyama received the B.E. and M.E. degrees in electrical engineering and the Dr. degree in information science from Waseda University, Japan, in 1991, 1993, and 1999, respectively. He is an associated professor of the Department of Computer Science at College of Information and Engineering, Ritsumeikan University. He was a visiting researcher at Institute for Software Research (ISR) of University of California, Irvine (UCI). His research interests include software refactoring, program analysis, software reuse, object-oriented design and programming, and software development environments. He is a member of the IEEE Computer Society, ACM, IPSJ, and JSSST.

Shinichiro Yamamoto is an associate professor of Information Science and Technology at Aichi Prefectural University. He received D.E. from Nagoya University in 1995. He is interested in formal method, program verification, and software develop environments. He is a member of IPSJ and JSSST.