Cross-Language Functional Testing for Middleware

A. Puder[1] and L. Wang[2]

[1] San Francisco State University
Computer Science Department
1600 Holloway Avenue
San Francisco, CA 94132
arno@sfsu.edu
[2] Computer Science Department
University of Southern California
941 W. 37th Place
Los Angeles, CA 90089-0781
limeiwan@usc.edu

Abstract. Middleware is at the heart of any distributed application and its correctness therefore requires rigorous testing. Since middleware technologies typically support heterogeneous environments, its API is available for different programming languages. Functional tests written to test the functionality of a middleware platform therefore have to be rewritten for all those programming languages. The framework introduced in this paper shows how functional tests written in Java can automatically be translated to other programming languages such as C++. This is achieved by using the XML-based programming language XMLVM. XMLVM can automatically be created from Java class files. The cross-language translations are accomplished by using XSL-transformations of XMLVM programs.

1 Motivation

Middleware allows the development of cross-platform, language-independent, distributed applications. Middleware is used in different contexts such as eCommerce applications or system-to-system integration, which places high demand on the correctness of middleware platforms. Several activities have created tens of thousands of test cases to ensure the correct behavior of a middleware technology. Among those efforts are CORVAL, COST, and WS-I (see [2], [15], and [4] respectively). One of the challenges of middleware functional testing is that by definition a middleware platform supports multiple programming languages. The implication of this heterogeneity is that functional tests have to be written in every language that is supported by a middleware technology, which leads to redundant and error-prone work.

Of the 100,000 lines of functional tests that were contributed as part of the COST (CORBA Open Source Testing, see [15]) effort, roughly half of the code tests C++ API whereas the other half tests the Java API of CORBA implementations. Every test therefore exists in two different implementations: C++ and Java. While these tests are functionally identical, they have to be re-written because of different language mappings for C++ and Java. The framework introduced in this paper allows a functional test to be written in Java and then to automatically derive the same test for other programming languages. This reduces manual work as well as potential for errors. We achieve this goal by making use of some advanced XML technologies. At the core of our framework is an XML-based programming language that allows cross-language translation of functional tests written in Java.

The paper is structured as follows: Section 2 introduces the problems related to writing functional tests for two different middleware technologies: CORBA and Web Services. Section 3 describes our framework. We present our XML-based programming language XMLVM and show how functional tests written in Java can automatically be translated to C++. Section 4 provides a conclusion and outlook.

2 Functional Testing of Middleware

This section highlights two real-life examples of functional testing for two different middleware technologies: CORBA and Web Services. In both cases it will become evident that a lot of testing code has to be virtually replicated in every programming language that is supported by the respective middleware technology.

2.1 Use case 1: CORBA Functional Testing

CORBA (Common Object Request Broker Architecture) defines an architecture for a platform independent middleware for object-oriented applications (see [1]). The core specification of CORBA, as standardized by the OMG, consists of over 1000 pages with hundreds of API functions. A functional test written for any of those functions would need to be translated into all languages that are being supported by CORBA. The following example illustrates this problem. The Java code excerpt demonstrates a functional test for a *Dynamic Any* using JUnit (see [9]):

```
J1:  // Java
J2:  public class DynAnyBaseTest extends junit.framework.TestCase {
J3:
J4:      private org.omg.CORBA.ORB orb = null;
J5:      private org.omg.DynamicAny.DynAnyFactory dynany_factory = null;
J6:
J7:      // ...
J8:
 J9:      public void testAccessBasicValue ()
J10:      {
J11:          int                        longVal1;
J12:          int                        longVal2;
J13:          org.omg.CORBA.TypeCode     tc = null;
J14:          org.omg.DynamicAny.DynAny  dynAny = null;
J15:
J16:          longVal1 = 700;
J17:          longVal2 = 0;
J18:          tc = orb.get_primitive_tc (org.omg.CORBA.TCKind.tk_long);
J19:          dynAny = dynany_factory.create_dyn_any_from_type_code (tc);
J20:          dynAny.insert_long (longVal1);
J21:          longVal2 = dynAny.get_long ();
J22:          assertEquals ("DynamicAny error", longVal1, longVal2);
J23:      }
J24:  }
```

The code above was taken from an actual functional test from the COST project. A Dynamic Any is a generic container for one data item. The type of the data item that can be contained in the Dynamic Any is determined when the Dynamic Any is created (lines J18 and J19). The Dynamic Any supports all types of the CORBA-IDL. The example above shows a simple functional test that first writes a `long` with value 700 into a Dynamic Any (line J20), extracts the value contained in the Dynamic Any (line J21) and then compares the two values to make sure that they are identical (line J22). Below is the same functional test, but now written in C++ using CPPUnit (see [8]):

```
 C1:  // C++
 C2:  class DynAnyBaseTest : public CppUnit::TestCase {
 C3:   private:
 C4:      CORBA::ORB_ptr                 orb;
 C5:      DynamicAny::DynAnyFactory_ptr  dynany_factory;
 C6:
 C7:      // ...
 C8:
 C9:   public:
C10:      void testAccessBasicValue()
C11:      {
C12:          CORBA::Long             longVal1;
C13:          CORBA::Long             longVal2;
C14:          CORBA::TypeCode_var     tc;
C15:          DynamicAny::DynAny_var  dynAny;
C16:
C17:          longVal1 = 700;
C18:          longVal2 = 0;
C19:          tc = CORBA::TypeCode::_duplicate (CORBA::_tc_long);
C20:          dynAny = dynany_factory->create_dyn_any_from_type_code (tc);
C21:          dynAny->insert_long (longVal1);
C22:         longVal2 = dynAny->get_long ();
C23:         CPPUNIT_ASSERT_EQUAL_MESSAGE ("DynamicAny error",
C24:                                       longVal1, longVal2);
C25:     }
C26: };
```

Conceptually the functional test above is doing exactly the same as the Java version, except that the CORBA's C++ API and CPPUnit are used in this case. Despite the similarities there are some differences. E.g., the way the `TypeCode` is created (J18 vs. C19) or how to use the assert-API in JUnit and CPPUnit (J22 vs. C23).

2.2 Use case 2: Web Services

Web Services are an emerging technology that have made a lot of head-waves over the past few years. Conceptually identical to CORBA, it has gained certain prominence because of Microsoft's commitment to support Web Services. XML is used extensively as the underlying foundation of many of the Web Services standards. The WS-I (Web Services Interoperability) organization issues the set of standards (called basic profile) that define the scope of Web Services (see [4]). It is interesting to note that Web Services do not support the concept of portability. I.e., the API for a certain programming language might differ significantly between different Web Services products.

The following code excerpt illustrates this problem:

```
// Java using Sun's WS Developer Kit
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface AccountIF extends Remote {
    public void deposit (int amount) throws RemoteException;
    public void withdraw (int amount) throws RemoteException;
    public int balance () throws RemoteException;
}
```

The code excerpt above shows the server-side mapping of a simple bank account interface using Sun's Web Services SDK. In Sun's implementation of the Web Services standards, the server-side implementation must implement a Java interface that extends the `Remote` interface. Furthermore, every method that belongs to the interface must throw the exception `RemoteException`. Below is a code excerpt that shows the same bank account interface using BEA's WebLogic Server:

```
// Java using BEA's Web Logic Server
public class Account implements com.bea.jws.WebService
{
    static final long serialVersionUID = 1L;
```

```
/**
 * @common:operation
 */
public void deposit (int amount);
{
    ...
```

As can be seen, BEA implements the bank account interface through a Java class that is derived from `com.bea.jws.WebService`. All remote methods are marked through a special JavaDoc comment `@common:operation`. It is apparent that functional testing for Web Services pose even greater challenges than for CORBA. Due to lack of portability, the functional tests have to be re-written for each Web Service product, even though Java is used in all instances.

3 Framework

This section introduces our framework. The goal is to write a functional test only once, and create the functional test for different languages automatically. In Section 3.1 we briefly discuss a non-solution. Section 3.2 gives an overview of Java's virtual machine. Based on those explanations, we introduce our XML-based programming language called XMLVM in Section 3.3. Section 3.4 finally describes how we use XMLVM to solve the problem of cross-language functional testing.

3.1 Non-solution

Before we present our solution to cross-language functional testing, we want to briefly discuss a non-solution. Initially we took the approach of defining a new programming language based on XML. Flow control statements (such as `if` and `while`) and other elements of an object-oriented programming language are represented by appropriate XML-tags. There are numerous projects that have created such XML-based programming languages (see [5, 12, 13, 7, 11]). Once a test case has been written in this language, it is relatively easy to translate it to another high-level programming language such as Java or C++. This can easily be accomplished by using XSL-transformations (see Figure 1).

However once we started to pursue this idea, we quickly realized that there were several disadvantages over using an XML-based programming language in this way. First and foremost programmers have to learn a new programming language. Someone using this approach would need to master a new programming language for which no tools (such as smart editors or syntax checkers) exist. Another problem resulted in the fact that XML tends to be very "verbose". By this we mean that it takes on the average more lines of code to express an algorithm in XML compared to other high-level languages. This is because of the rigid syntax that XML imposes on the structure of a document. For these reasons we took a different approach.

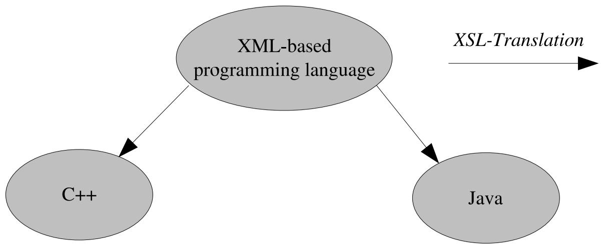

Fig. 1. Non-solution: Mapping XML to C++ and Java.

3.2 Java's Virtual Machine

As outlined in the previous section, it is not practical to expose a programmer to an XML-based programming language. Yet XML has much to offer due to the availability of rich tool sets. In order to exploit the benefits of XML, but make it transparent to programmers, we created a low-level XML-based programming language that is not intended for human readers. In order to use standards as much as possible, we decided to use the byte code executed by a Java virtual machine as a model for our XML-based programming language.

Before we explain our approach, we provide a few details on the Java Virtual Machine concept (for more details see [10]). A Java compiler translates the Java source code to hardware independent byte code that is stored in a class file. The byte code resembles the machine code of other hardware architectures. The Java virtual machine implements a simple stack based machine (see Figure 2).

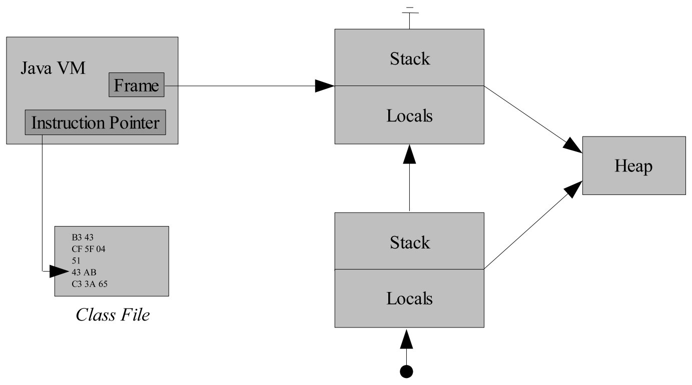

Fig. 2. Java VM.

The Java VM maintains an instruction pointer to the class file that points to the next instruction to be executed. Upon entering a method, a new frame consisting of a stack and local variables is created. This frame will be deleted upon exiting the method. The Java VM maintains a pointer to the current frame (which represents the most nested method call). A method has only access to its own stack and local variables. The actual parameters of a method are automatically stored in the local variables. Besides the stack frames, the Java VM maintains a garbage collected heap where a program can allocate new objects.

The Java byte code features a mix of low-level and high-level virtual machine instructions. On the one hand side one finds simple instructions such as `iadd` that pops two integers off the stack and pushes the sum back onto the stack. On the other side there exist high-level instructions such as `new` (for instantiating new objects) and `invokevirtual` (invoke a virtual method). These instructions go beyond the capabilities of normal machine languages and explain the difficulties in creating a real CPU that can execute Java byte code natively.

3.3 XMLVM

The Java byte code resembles the machine code of other hardware architectures. It is interesting to note that Sun Microsystems as the inventor of Java never standardized an assembly language of their own byte code. Several assemblers were developed, but they had to invent their own syntax. The most commonly used assembly syntax stems from the Jasmin project (see [14]).

The first step in creating a cross-language functional testing framework consist in defining an XML-based programming language that is based on the Java byte code. This effectively defines an assembly language for the Java virtual machine whose syntax is based on XML. Since we mimic the syntax very closely to the Java byte code instructions, there is a direct bi-directional mapping between Java class files and our XML-based programming language. Since the XML-based programming language is closely related to the byte code of the Java virtual machine, we call our language XMLVM.

In practice, programs are not directly written in XMLVM, but rather created automatically from class files. The programmer is thus not exposed to the details of XMLVM, but can implement his or her programs in Java. The following XML follows the XMLVM schema and demonstrates the translation of the Java functional test for Dynamic Any presented in Section 2.1:

```
 X1:  <?xml version="1.0" encoding="UTF-8"?>
 X2:  <xmlvm>
 X3:    <class name="DynAnyBaseTest" isPublic="true"
 X4:           isSynchronized="true" extends="junit.framework.TestCase">
 X5:      <field isPrivate="true" name="orb"
 X6:             type="org.omg.CORBA.ORB" />
 X7:      <!-- ... -->
 X8:      <method name="testAccessBasicValue"
 X9:              isPublic="true" stack="3" locals="5">
X10:        <signature>
X11:          <return type="void" />
X12:        </signature>
```

```
X13:        <code>
X14:          <!-- ... -->
X15:          <getfield class-type="DynAnyBaseTest"
X16:                    type="org.omg.CORBA.ORB" field="orb" />
X17:          <getstatic class-type="org.omg.CORBA.TCKind"
X18:                     type="org.omg.CORBA.TCKind" field="tk_long" />
X19:          <invokevirtual class-type="org.omg.CORBA.ORB"
X20:                         method="get_primitive_tc">
X21:            <signature>
X22:              <return type="org.omg.CORBA.TypeCode" />
X23:              <parameter type="org.omg.CORBA.TCKind" />
X24:            </signature>
X25:          </invokevirtual>
X26:          <astore type="java.lang.Object" index="3" />
X27:          <!-- ... -->
X28:        </code>
X29:      </method>
X30:    </class>
X31:  </xmlvm>
```

The above XML was automatically created using our tool. The complete XML is much longer and cannot be reproduced here. A few details are worth mentioning. We will relate the XMLVM output with the original Java functional test from Section 2.1. Line X3 contains the class declaration (line J2). Line X5 contains the instance member `orb` (line J4). Line X8 contains the declaration of method `testAccessBasicValue()` (line J9). The `stack` and `locals` attributes in line X9 state how big the stack and how many local variables are needed for this method. Note that the Java compiler computes this information by doing a flow analysis. Lines X10 to X12 show the signature, and lines X13 to X28 the implementation of method `testAccessBasicValue()`.

Lines X15 through X26 show an excerpt of the byte code generated by the Java compiler. Those lines represent the compiled version of Java source code at line J18. There are basically four byte code instructions: <`getfield`> (line X15) pushes the value of instance member `orb` onto the stack and <`getstatic`> (line X17) pushes the value of static variable `org.omg.CORBA.TCKind.tk_long` onto the stack. <`invokevirtual`> (line X19) calls the virtual method `get_primitive_tc()`. This instruction assumes that the object reference to the target object as well as the actual parameters are on the top of the stack (which was done by the previous two instructions). Once the call to `get_primitive_tc()` returns, the result is on the top of the stack. <`astore`> (line X26) pops this result off the stack and saves it in a local variable.

3.4 Mapping XMLVM to other languages

The XML presented in the previous section was automatically generated and it represents an intermediate artifact not intended to be inspected by programmers. The principal idea of our framework is to translate XMLVM to other high-level programming languages. The translation is done using XSL-translations (see [3]). Figure 3 shows the overall translation process.

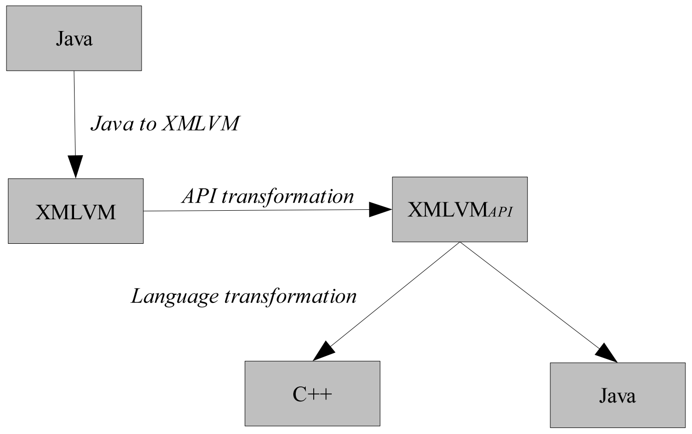

Fig. 3. XMLVM transformation process.

API transformation: As shown in the figure, the source program is first translated to XMLVM by a tool. The resulting XMLVM then undergoes an API transformation. The purpose of the API transformation is to adapt the API from the source to the target language. The original test case was written using specific APIs such as Java-CORBA and JUnit. If the test case is to be translated to C++, this API has to be adapted to the appropriate API available for the target language. For each API exist a XSL-stylesheet that adapts the API. The following list gives some examples of API transformations if the target language is C++:

– JUnit to CPPUnit:
  - Base class `junit.framework.TestCase` (J2) changes to `CppUnit.TestCase` (C2)
  - `assertEquals()` (J22) changes to `CPPUNIT_ASSERT_EQUAL_MESSAGE()` (C23)

– CORBA for Java to C++:
  - Namespace prefix `omg.org.CORBA` (J4) changes to `CORBA` (C4)
  - Method `get_primitive_tc()` (J18) changes to `_duplicate()` (C19)

Note that the output of API transformation is still an XMLVM program (referred to as XMLVM$_{API}$ in Figure 3) and consequently uses XMLVM notation, such as “`.`” for the scope operator. The examples given above are mostly simple renaming operations that can easily be achieved by appropriate XSL-stylesheets.

The more complex example is the creation of a `TypeCode` (J18). This situation can be handled by a more complex XSL-stylesheet. This XSL-template basically looks for a call to `get_primitive_tc()` and then transforms the API to a semantically equivalent version to be used for C++.

Each API transformation is handled by its own XSL-stylesheet and depending on how many different libraries (e.g., JUnit, CORBA, etc) are used, multiple stylesheets may be applied. The result of the API transformation is again an XMLVM program. The excerpt below demonstrates the resulting XMLVM after the applying the XSL-stylesheets for API transformation:

```
 A1:  <getstatic class-type="CORBA"
 A2:             type="CORBA.TypeCode" field="_tc_long" />
 A3:  <invokestatic class-type="CORBA.TypeCode"
 A4:                method="_duplicate">
 A5:    <signature>
 A6:      <return type="CORBA.TypeCode" />
 A7:      <parameter type="CORBA.TypeCode" />
 A8:    </signature>
 A9:  </invokestatic>
A10:  <astore type="CORBA.Object" index="3" />
```

The excerpt above shows the result of translating the original XMLVM code for creating a `TypeCode` (lines X15 to X26 in Section 3.3). Instead of using the ORB-singleton to create a `TypeCode` via `get_primitive_tc()`, the `TypeCode` is now created by duplicating the constant `_tc_long` that all CORBA conformant C++ ORBs are required to have.

Language transformation: The result of the API transformation is another XMLVM program. The final step in this translation process consists in generating code for the target language. This translation is done by yet another XSL-stylesheet. The idea for this last step of our framework is to map XMLVM-instructions one-to-one to the target language, without attempting to reverse engineer (or de-compile) the original Java program. Since the Java VM is based on a simple stack-based machine, we simply mimic a stack-machine in the target language. An example helps to illustrate this approach. The XMLVM instruction <`astore`> pops an object reference off the stack and saves it to a local variable. Here is the XSL-template that creates C++ code for this instruction:

```
<xsl:template match="astore">
    <xsl:text>
    locals[</xsl:text>
    <xsl:value-of select="@index"/>
    <xsl:text>] = stack.pop();</xsl:text>
</xsl:template>
```

As an example, the <`astore`> instruction in line A10 would translate to the following C++ code:

```
locals[3] = stack.pop();
```

This C++ code makes reference to variables `locals` and `stack`. Those variables are declared for every method and it is with the help of those variables that we mimic the VM's stack-machine. The code below represents the C++ version of the XMLVM program shown in lines A1 to A10 of the previous section:

```
 T1:  // C++
 T2:  class DynAnyBaseTest
 T3:          : public virtual CppUnit::TestCase
 T4:  {
 T5:      CORBA::ORB_ptr orb;
 T6:      DynamicAny::DynAnyFactory_ptr dynany_factory;
 T7:
 T8:      // ...
 T9:
T10:      void testAccessBasicValue()
T11:      {
T12:          XMLVM::Locals locals(5);
T13:          XMLVM::Stack stack(3);
T14:          XMLVM::Object op1;
T15:          XMLVM::Object op2;
T16:          locals[0] = this;
T17:
T18:          // ...
T19:          stack.push(CORBA::_tc_long);
T20:          op1 = CORBA::TypeCode::_duplicate((CORBA::TypeCode_ptr)
T21:                                            stack.top(0));
T22:          stack.remove(1);
T23:          stack.push(op1);
T24:          locals[3] = stack.pop();
T25:          // ...
T26:      }
T27:  }
```

As can be seen from the code excerpt, there is a natural mapping from XMLVM to C++. The intention is not to generate readable code, but correct code that uses the API of the target language. The above code is automatically created by the XSL-language transformation and is not meant to be inspected by programmers. We mimic Java's VM via the two classes `XMLVM::Locals` and `XMLVM::Stack` (lines T12 and T13). Those two C++ classes are part of the XMLVM library for C++. Class `XMLVM::Stack` features common stack-operations such as push and pop. Both of these classes implement the garbage collection that is normally done by Java's VM. Variables `op1` and `op2` (lines T14 and T15) are used as temporary variables needed by some XMLVM-instructions.

4 Conclusions and Outlook

Functional testing for middleware requires individual tests to be re-written in all programming languages that are supported by that middleware. The framework introduced in this paper proposes a novel way to automate this manual and error prone task. In our framework functional tests are written once in Java. The class file that contains the compiled version of the functional test is then translated to XMLVM; an XML-based programming language. Then various XSL-transformations can be applied to first transform the API and then to translate the functional test to another high level language.

It is important to emphasize the fact that we effectively translate functional tests written in Java to other programming languages. This works well for APIs (such as Dynamic Any) that exists in all different languages, but there are limitations for language specific APIs. E.g., the CORBA C++ language mapping defines various helper types for C++ pointers that can be recognized by the suffix `_ptr` and `_var`. These helper types do not exist in Java, simply because C++ pointers are much more complex than Java object references. Functional tests that specifically test the correctness of these helper types would need to be written manually in C++.

XMLVM is at the core of our framework. We have implemented it based on the Byte Code Engineering Library (BCEL) which is part of Apache's Jakarta project (see [6]). We are currently investigating other uses of XMLVM in different contexts. One possible use could be in a code migration framework for web-based applications based on previous work (see [16]). Another potential use of XMLVM could be byte code instrumentation using XSL-transformation.

References

1. Common object request broker architecture (corba/iiop). Object Management Group. http://www.omg.org/technology/documents.
2. Vsorb test suite specification, release 1.0.0. Open Group, 1997. http://www.opengroup.org/corval/vsorbts.pdf.
3. Xsl transformations. World Wide Web Consortium (W3C), 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.
4. Web Services Interoperability Organization, 2004. http://www.ws-i.org.
5. G.J. Badros. A markup language for java source code. May 2000. http://www.cs.washington.edu/research/constraints/web/badros-javaml-www9.pdf.
6. Markus Dahm. Byte code engineering. Java Informations Tage, pages 267–277, 1999.
7. W. Emmerich, C. Mascolo, and A. Finkelstein. Implementing incremental code migration with xml. pages 397–406. In M. Jazayeri and A. Wolf, editors, Proc. 22nd Int. Conf. on Software Engineering (ICSE2000) Limerick, Ireland, ACM Press., June 2000. http://www.cs.ucl.ac.uk/staff/W.Emmerich/publications/ICSE2000/MobXML/mobxml.pdf.
8. Paul Hamill. *Unit Test Frameworks*. O'Reilly; 1 edition, October 2004.

9. Andy Hunt and Dave Tomas. *Pragmatic Unit Testing in Java With JUnit.* The Pragmatic Programmers; 1 edition, September 2003.
10. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley Pub Co, second edition, April 1999.
11. Jonathan I. Maletic, Michael L. Collard, and Andrian Marcus. Source code files as structured documents. pages 289–292, June 2002. http://www.sdml.info/papers/iwpc02.pdf.
12. E. Mamas and K. Kontogiannis. Towards portable source code representation using xml. pages 172–182. Proceedings of the Seventh Working Conference on Reverse Engineering, IEEE Computer Society Press, Brisbane Australia, November 2000.
13. G. McArthur, J. Mylopoulos, and S.K.K. Ng. An extensible tool for source code representation using xml. pages 199–208. Proceedings of the Ninth Working Conference on Reverse Engineering, IEEE Computer Society, Richmond, Virginia, USA, October 2002.
14. Jonathan Meyer. An assembler for the java virtual machine. 1996. http://jasmin.sourceforge.net/.
15. Arno Puder. Corba open source testing. OMG in Motion, Needham, 2001.
16. Arno Puder. Extending desktop applications to the web. Second Workshop on Distributed Objects Research, Experiences and Applications (DOREA 2004), pages 8–11, Dublin, July 2004. Computer Science Press.