XMLVM User Manual

Contents

Chapter 1

Introduction

XMLVM is a flexible cross-compilation framework. Instead of cross-compiling source code of high-level programming languages, XMLVM translates byte code instructions. Byte code instructions are represented by XML-tags and the cross-compilation is done via XSL stylesheets. This chapter gives an introduction to XMLVM. Section 1.1 provides a brief overview of the XMLVM toolchain. Section 1.2 describes how to obtain the source code of XMLVM and Section 1.3 how to build XMLVM from source. The various command line options supported by XMLVM are described in Section 1.4.

1.1 Overview

XMLVM supports byte code instructions from two popular virtual machines: the Java Virtual Machine (JVM) and the Common Language Runtime (CLR) that is part of the .NET framework. The name XMLVM is inspired by the fact that byte code instructions are represented via XML. Each byte code instruction is mapped to a corresponding XML-tag. Transformations of XMLVM programs are done via XSL stylesheets. Figure 1.1 shows all possible paths through the XMLVM toolchain.

The first step in using XMLVM is to compile a Java or .NET source code program to byte code. This is done with a native compiler such as Sun Microsystems `javac` or Microsofts Visual Studio. The resulting byte code program (either a Java `.class` file or a .NET `.exe` file) is fed into the XMLVM toolchain where it is first converted to XML. XMLVM_{JVM} denotes an XMLVM program that contains JVM byte code instructions, whereas a XMLVM_{CLR} program contains CLR byte code instructions. It is possible to cross-compile XMLVM_{CLR} to XMLVM_{JVM} with the help of a data flow

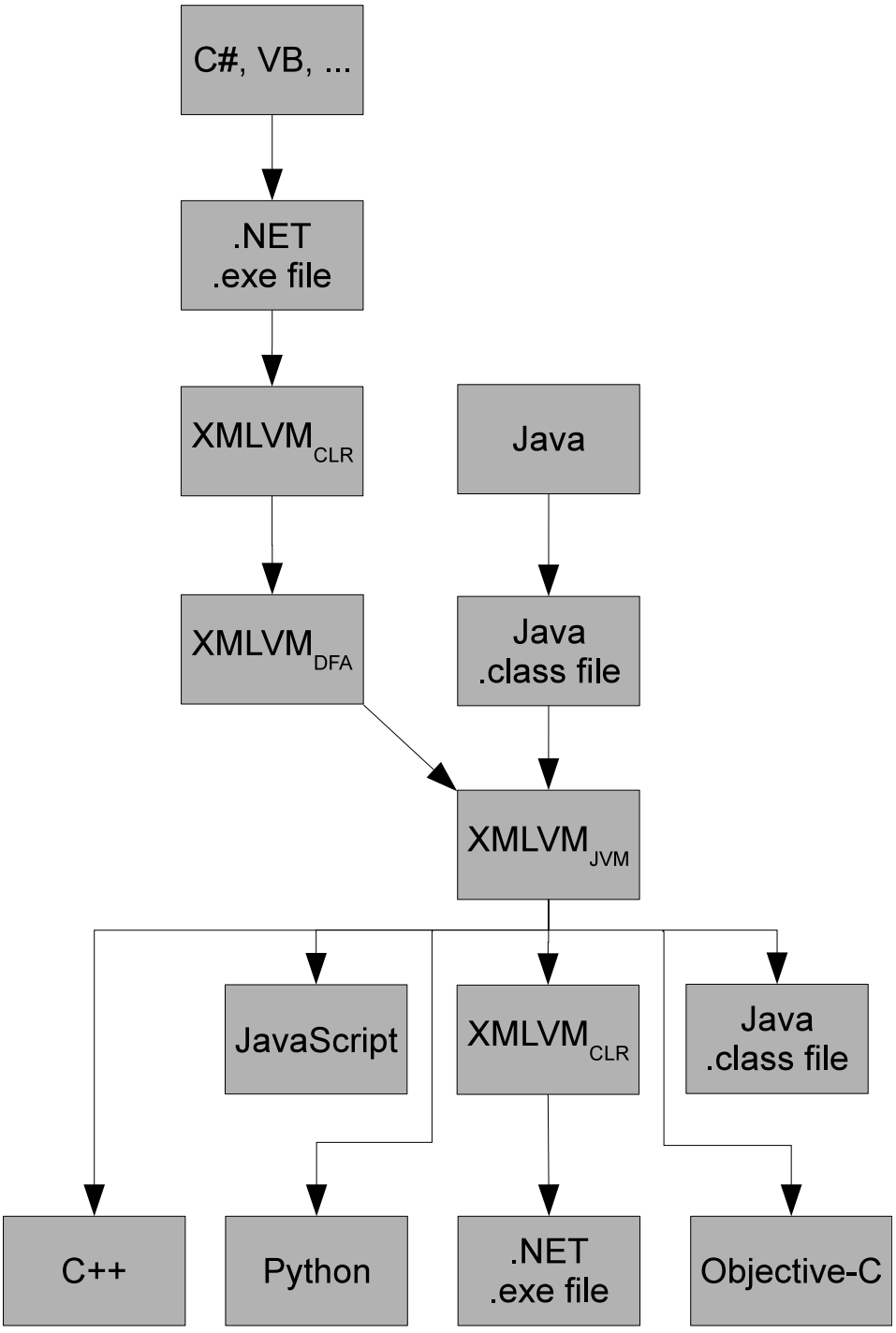

Figure 1.1: XMLVM Toolchain.

analysis (denoted as XMLVM_{DFA} in Figure 1.1).

XMLVM_{JVM} serves as the canonical representation within the XMLVM toolchain in the sense that it separates the frontend from the backend. That is to say, all code generating backends use XMLVM_{JVM} as their input. As can be seen in Figure 1.1, various paths through the XMLVM toolchain are possible. For example, .NET programs can be cross-compiled to Java class files and Java class files can be cross-compiled to JavaScript amongst others.

Table 1.1: Completeness of various XMLVM backends.

To:	**From:**	
	JVM	**CLR**
C++	Language cross-compilation only. No library support.	Language cross-compilation only. No library support.
JavaScript	Compatibility library for a subset of AWT.	Compatibility library for a subset of WinForms.
Python	Language cross-compilation only. No library support.	Language cross-compilation only. No library support.
.NET	Language cross-compilation only for a subset of JVM instructions.	N/A
Java	N/A	Support for most .NET instructions. No support for generics. Compatibility library for a subset of WinForms.
Objective-C	Most of language cross-compilation. Compatibility libraries for a subset of Cocoa.	Language cross-compilation only. No library support.

The byte code level cross-compilation is only one aspect of XMLVM. The XMLVM distribution also contains compatibility libraries for the various targets. For example, When cross-compiling from C# to Java class files, XMLVM contains a compatibility library for WinForms (the Microsoft GUI library) written in Java. This allows C# desktop applications to be cross-compiled to Java desktop applications. Similarly, when cross-

compiling from Java to JavaScript, XMLVM features a compatibility library for AWT/Swing written in JavaScript that effectively allow to cross-compile Java desktop applications to AJAX applications.

It should be noted that XMLVM is a research project and as such lacks the completeness of a commercial product. Each individual backend requires a significant effort to support different API. WinForms, AWT/Swing, and Cocoa are all complex libraries and at this point XMLVM only supports a subset of each. The various paths through the XMLVM toolchain have different levels of maturity that should be taken into consideration when using XMLVM. Table 1.1 gives an overview of the completeness of the various backends. An in-depth overview of the theoretical foundations of XMLVM can be found in [1].

1.2 Getting XMLVM

XMLVM is released under the GPL v2 license and is hosted at SourceForge. We currently do not offer pre-compiled binary packages. The only way to obtain XMLVM is to checkout the latest version from the Subversion repository. You will need a Subversion client to do this. If you are using a command line version of Subversion, you can checkout the trunk of the XMLVM repository via the following command:

```
svn co https://xmlvm.svn.sourceforge.net/svnroot/xmlvm/trunk/xmlvm
```

Note that this will give you a read-only version of the repository. You will be able to update (which you should do frequently) but not commit changes to the repository. If you find a bug, please send a mail to the XMLVM mailing list.

XMLVM is developed using the Eclipse IDE. You can also checkout the sources of XMLVM via Eclipse (using an appropriate Subversion plugin such as Subclipse or Subversive). The XMLVM sources contain `.project` and `.classpath` files so that Eclipse will recognize XMLVM as an Eclipse project. The benefit of using Eclipse is that it makes it easy to navigate the source code if you intend to study the internals of XMLVM. There are also numerous Eclipse launch configurations (in the `etc/` directory) that allow the invocation of various demos.

1.3 Compiling XMLVM

XMLVM depends on numerous third-party libraries such as BCEL, JDOM, or Saxon. All these libraries are also released under an Open Source library. To facilitate the compilation process, XMLVM contains binary versions (i.e., jars) of all required libraries. All third-party libraries are contained in the `lib/` directory. Building XMLVM from sources requires Java 1.6 as well as ant. In order to compile XMLVM from command line, simply run ant in the XMLVM root directory:

```
cd xmlvm
ant
```

After a successful run of ant, there should be a `dist/` directory. The ant script packages all dependent libraries and XMLVM's own class files into one jar file. The only file needed to run XMLVM is the jar file `dist/xmlvm.jar`. This jar file can be copied to a convenient location. The following section explains how to run XMLVM. The directory `dist/demo/` contains several demos to highlight the various aspects of XMLVM.

1.4 Invoking XMLVM

As mentioned in the previous section, the ant script will package the binaries of XMLVM into one jar file. By default, this jar file is located in `dist/xmlvm.jar` after a successful compilation of XMLVM. Java 1.6 is needed to run XMLVM. Invoking XMLVM can be done in the following way:

```
java -jar dist/xmlvm.jar
```

Command line options can be appended at the end of the command line such as:

```
java -jar dist/xmlvm.jar --version
```

The various byte code transformations and code generators can be invoked via appropriate command line options. Section 1.4.1 explains all available command line options and Section 1.4.2 gives some examples. Note that at this point we only give an overview of the command line options. Refer to subsequent chapters for more detailed information on the various backends.

1.4.1 Command Line Options

XMLVM can be invoked by running the executable jar file called `xmlvm.jar`. In the following we assume that an alias called `xmlvm` is defined to invoke XMLVM. Under Unix, this can be accomplished via the following command:

```
alias xmlvm="java -jar dist/xmlvm.jar"
```

The behavior of XMLVM is controlled by numerous command line arguments. `xmlvm` reads in one or more source files, processes them according to the command line options, and then writes out one or more destination files.

`--in=<path>`

The source files are specified via one or more `--in` options. If the argument passed to `--in` is a directory, then this directory is traversed recursively and all files with the suffix `.class`, `.exe`, or `.xmlvm` are processed. Files with other suffixes are ignored. It is possible to use wildcards to filter out certain files. It is possible to specify multiple `--in` parameters. At least one `--in` parameter is required.

`--out=<path>`

The output generated by `xmlvm` is written to a directory specified by the `--out` parameter. The argument `<path>` has to denote a directory name. If the directory does not exist, `xmlvm` will create it. All files generated by `xmlvm` will be written to this directory. The only exception is when using `--target=class`. In this case the resulting Java class files (ending in suffix `.class`) are written to appropriate sub-directories matching their package names. Already existing files with the same name will be overwritten. If the `--out` parameter is omitted, the current directory is the default.

`--target=[xmlvm|jvm|clr|dfa|class|exe|js|cpp|python|objc|iphone]`

This option defines the output format of the target. These correspond with the various backends for code generation supported by XMLVM. The different targets are explained in the following:

`xmlvm:` The input files are cross-compiled to XMLVM. `*.class` files will be cross-compiled to XMLVM$_{JVM}$. `*.exe` files will be cross-compiled to XMLVM$_{CLR}$. `*.xmlvm` files will be copied unchanged. This option is the default for `--target`.

`jvm`: The input files are cross-compiled to XMLVM$_{JVM}$.

`clr`: The input files are cross-compiled to XMLVM$_{CLR}$

`dfa`: A DFA (Data Flow Analysis) is performed on the input files. Currently the DFA will only be performed for XMLVM$_{CLR}$ programs. This option cannot be used in conjunction with any other code generating option.

`class`: The input files are cross-compiled to Java class files.

`exe`: The input files are cross-compiled to a .NET executable.

`js`: The input files are cross-compiled to JavaScript.

`cpp`: The input files are cross-compiled to C++.

`python`: The input files are cross-compiled to Python.

`objc`: The input files are cross-compiled to Objective-C.

`iphone`: Cross-compiles an application to the iPhone. The output directory specified by `--out` will contain a ready to compile iPhone application. The resulting iPhone application can be compiled via "make" using Apple's SDK for the iPhone. This option requires the option `--iphone-app`.

`--iphone-app=<app_name>`

This option can only be used in conjunction with option `--target=iphone`. It specifies the name of the iPhone application whose name will be `<app_name>`.

`--android2iphone`

Cross-compiles an Android application to the iPhone. This option requires `--target=iphone`..

`--qx-app=<app_name>`

Cross-compiles an application to a Qooxdoo application. The environment variable `QOOXDOO_HOME` needs to point to the base directory of the Qooxdoo installation. The application will be called `<app_name>`. The output directory specified by `--out` will contain a ready to run Qooxdoo application. This option implies `--target=js` and requires option `--qx-main`.

`--qx-main=<main-class>`

This option denotes the entry point of the generated Qooxdoo application. It requires a full qualified name as a parameter. This option can only be used in conjunction with option `--qx-app`.

`--qx-debug`

Creates a debug version of the Qooxdoo application. If not specified, a ready-to-deploy version will be generated. Requires option `--qx-app`.

`--version`

Prints the version of XMLVM.

`--quiet`

No diagnostic messages are printed.

1.4.2 Examples

`xmlvm --in=/foo/bar`

The directory `/foo/bar` is searched recursively for `*.class`, `*.exe`, and `*.xmlvm` files. The default target is `xmlvm`. For `*.class` files, XMLVM_{JVM} is generated. For `*.exe` files, XMLVM_{CLR} is generated. Files with suffix `*.xmlvm` are copied to the output directory. Other files with different suffices are ignored. Since no `--out` parameter was given, the default output directory is "." (the current directory).

`xmlvm --in=/foo/*.class --in=/bar/*.exe --out=/bin`

The directory `/foo` is searched recursively for `*.class` and the directory `/bar` is searched recursively for `*.exe` files. The default target is `xmlvm`. Files with other suffices are ignored. For `*.class` files, XMLVM_{JVM} is generated. For `*.exe` files, XMLVM_{CLR} is generated. The resulting `*.xmlvm` files are placed in directory `/bin`.

`xmlvm --in=/foo --target=jvm`

The directory `/foo` is searched recursively for `*.class`, `*.exe`, and `*.xmlvm` files. In all cases, the generated output will always be XMLVM_{JVM}. For `*.exe` files as well as `*.xmlvm` files containing something other than XMLVM_{JVM} will be cross-compiled XMLVM_{JVM}.

`xmlvm --in=/foo --target=class`

Same as the previous example, however instead of generating XMLVM_{JVM} files, Java `*.class` files that can be executed by a Java virtual machine will be generated. The class files will be placed in appropriate sub-directories matching their package names.

`xmlvm --in=/foo --target=iphone --iphone-app=TheApplication`

Same as the previous example, however instead of creating Java `*.class` files, an iPhone application will be generated. The output directory will contain the ready to compile Objective-C source code including all necessary auxiliary files such as `Info.plist` and a `Makefile`. The iPhone application will be called `TheApplication` using a default icon.

`xmlvm --in=/foo --target=iphone --android2iphone --iphone-app=TheApplication`

Same as the previous example, but will also copy the Android compatibility libraries to the output directory. This effectively allows Java-based Android applications to be cross-compiled to the iPhone.

`xmlvm --in=/foo --qx-app=TheApplication --qx-main=com.acme.Main`

The directory `/foo` is searched recursively for `*.class`, `*.exe`, and `*.xmlvm` files. This option implies `--target=js`. All files will be cross-compiled to JavaScript. With the help of the Qooxdoo build scripts, the output directory will contain a ready to be deployed AJAX application. The main entry point of the application is `com.acme.Main`.

Chapter 2

iPhone/Android Backend

2.1 Generating Objective-C

2.2 Running an iPhone Application

2.2.1 Java-based iPhone Emulator

2.2.2 Apple's iPhone Emulator

2.2.3 Using Apple's Xcode IDE

2.3 Android Compatibility Library

Chapter 3

JavaScript/AJAX Backend

TBD

Appendix A

Frequently Asked Questions

Bibliography

[1] Arno Puder and Jessica Lee. Towards an XML-based Byte Code Level Transformation Framework. In *4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, York, UK, March 2009. Elsevier.