XMLVM User Manual

Note: the command line interface described in this document is not yet implemented. This note will be removed once the implementation is consistent with the documentation.

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

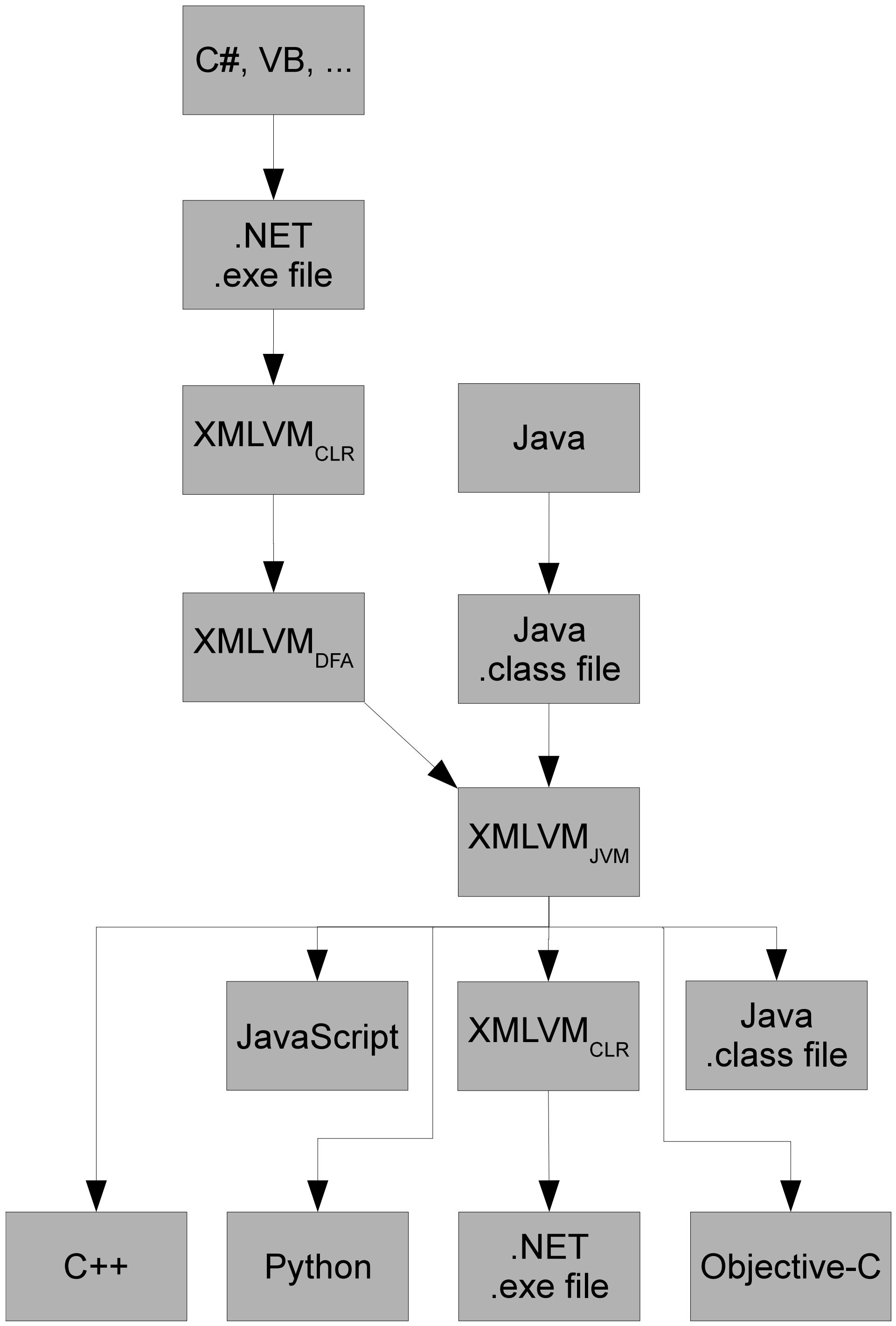

Figure 1.1: XMLVM Toolchain.

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|android-on-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`.

`android-on-iphone`**:** Cross-compiles an Android application to the iPhone. This option is mostly identical in behavior as target `iphone`, except that target `android-on-iphone` will also copy an Android compatibility library to the output directory specified by `--out`. This option also requires the option `--iphone-app`.

`--iphone-app=<app_name>`

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

`--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=android-on-iphone --iphone-app=TheApplication
```

Same as the previous example, but will also copy the Android compatibility library 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

With the help of XMLVM it is possible to cross-compile Java applications to native iPhone applications. The Apple license agreement does not permit the installation of a virtual machine on the iPhone. By cross-compiling a Java application to a native iPhone application, this restriction of the license agreement is therefore not violated. XMLVM can legally generate native iPhone application and it is not necessary to jailbreak a device in order to run an application cross-compiled by XMLVM. Section 2.1 gives an overview of Java-based iPhone applications including the obligatory "Hello World" program. Section 2.2 explains various ways how a Java-based iPhone application can be executed. Section 2.3 demonstrates how Android applications can be cross-compiled to native iPhone applications. Section 2.4 gives an overview of various sample applications that show the capabilities of the Java-for-iPhone portion of XMLVM.

2.1 Java-based iPhone Applications

Apple only supports Objective-C as the development language for the iPhone. The GUI of iPhone applications is based on Cocoa Touch. If Java is to be used as a development language for iPhone applications, two aspects need to be addressed: the cross-compilation of Java to Objective-C and a Java API for Cocoa Touch. The following application shows the "Hello World" application for the iPhone written in Java:

```
import org.xmlvm.iphone.*;

public class HelloWorld extends UIApplication {
```

```java

    public void applicationDidFinishLaunching(UIApplication app) {
        UIScreen screen = UIScreen.mainScreen();
        CGRect rect = screen.applicationFrame();
        UIWindow window = new UIWindow(rect);

        rect.origin.x = rect.origin.y = 0;
        UIView mainView = new UIView(rect);
        window.addSubview(mainView);

        UILabel title = new UILabel(rect);
        title.setText("Hello World!");
        title.setTextAlignment(UITextAlignment.UITextAlignmentCenter);
        mainView.addSubview(title);

        window.makeKeyAndVisible();
    }

    public static void main(String[] args) {
        UIApplication.main(args, HelloWorld.class);
    }

}
```

The Java API for Cocoa Touch is loosely based on the Objective-C counterpart. While XMLVM makes better use of overloading and interface definitions for delegates to create a strongly-typed API, the description of the various classes and methods can be taken from Apple's official documentation. Choosing the target `iphone` (see Section 1.4.1) will cross-compile this application to Objective-C, include all required compatibility libraries for Cocoa Touch, as well as generate a Makefile that facilitates the compilation of the native version of the application.

2.2 Running an iPhone Application

A Java-based iPhone application can by executed in several ways, each of which will be explained in the following.

2.2.1 Java-based iPhone Emulator

XMLVM includes Java implementations of the Cocoa Touch API. Classes such as `UIWindow` and `UILabel` have been implemented making use of Java2D. The implication is that a Java-based iPhone application can be run as a pure 100% Java application on a standard Java virtual machine without the need of Apple's SDK or an actual device. Note that re-implementing Cocoa Touch in Java is a major undertaking in itself and XMLVM only support a certain subset of the API. While this approach works well for the demos shipped with XMLVM (an online applet version of the demos can be viewed at `http://xmlvm.org`) there are currently no plans to make further enhancements of the Java-based iPhone emulator.

2.2.2 Apple's iPhone Emulator

Apple's SDK for the iPhone includes an iPhone emulator. When cross-compiling a Java application using the the `iphone` target, the resulting `Makefile` compiles and deploys the native application on Apple's emulator. The `Makefile` will only work on a Mac OS platform with a default installation of the Xcode toolchain. After cross-compiling an application using XMLVM, simply change to the directory where the code was generated and type "make". This will compile and run the application in Apple's emulator.

2.2.3 Using Apple's Xcode IDE

The `Makefile` generated by XMLVM can only compile and deploy the application on Apple's emulator. Given the complexity of code signing that Apple requires for all native iPhone applications, you will need to use Xcode if a cross-compiled Java application is to be deployed on a device. The following steps explain the process of compiling an XMLVM-generated application using Xcode:

1. First cross-compile the Java-based iPhone application using XMLVM by using target `iphone`.
2. Launch Xcode and create a new Cocoa application by selecting File > New Project... > Window Based Application.
3. Choose a name for the new project and a location where it will be stored. This location should not be the directory where XMLVM cross-compiled the application.
4. Xcode will automatically generate some skeleton files that need to be deleted. Delete `main.m`, `MainWindow.xib`, as well as the application delegate header and implementation file.
5. In `Info.plist`, delete the property "Main nib file base name."

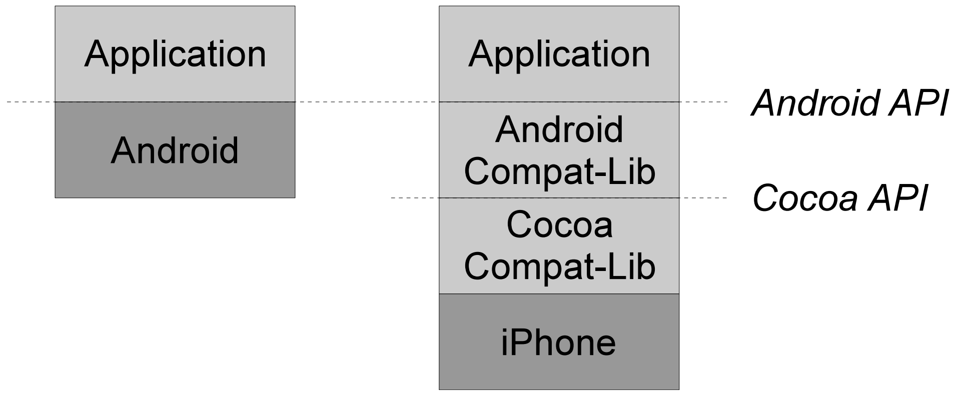

Figure 2.1: Android to iPhone cross-compilation.

6. Add all source files generated by XMLVM to the project by selecting Project > Add to Project... Do not add the `Info.plist`, `Makefile`, and `MakeVars` files that were generated by XMLVM.

7. Clicking on “Build and Go” should compile and launch the application in Apple’s emulator.

8. With the proper certificates and entitlements installed, you can change the target to a provisioned device and clicking on “Build and Go” now should compile and deploy the application on the device.

2.3 Android Compatibility Library

Android is an Open Source platform for mobile devices. Initiated by Google, Android has received much attention. Android applications are developed using Java, although a special compiler converts class files to a proprietary, regiser-based virtual machine that is used on Android devices to execute applications. Android defines its own API for writing mobile applications. With the help of XMLVM it is possible to cross-compile Java-based Android applications to native iPhone applications. Figure 2.1 depicts this process.

The Android application is written in Java and makes use of an Android specific API. XMLVM offers a compatibility library, written in Java, that offers the same API as Android, but only makes use of the Java-based API for Cocoa Touch mentioned earlier. During the cross-compilation process, both the application and the Android compatibility library are cross-compiled from Java to Objective-C and linked with the Cocoa Touch compatibility library to yield a native iPhone application.

As can be seen, compared to the Java-for-the-iPhone portion of XMLVM, the only additional feature added to support Android applications is the Android compatibility library. When selecting target `android-on-iphone` (see Section 1.4.1), additional to cross-compiling the application to Objective-C, this option also includes the Android compatibility library to the generated project. The Android-based iPhone application can be run in both the XMLVM-specific iPhone emulator as well as Apple's emulator as explained in the previous section.

2.4 Sample Applications

XMLVM includes several sample programs that demonstrate the Java-for-the-iPhone portion of XMLVM. All examples can be executed in one of the ways explained in Section 2.2: in XMLVM's own Java-based iPhone emulator, by running the `Makefile`, or by using the Xcode IDE. Additionally, the sample programs can also be run via Eclipse-specific launch configurations stored in the `etc/` directory. Those launch configurations will run the Java-based iPhone emulator that was described in Section 2.2.1.

All sample programs will get automatically compiled when running ant as explained in Section 1.3. After ant successfully completes, there will be a directory `dist/demo/`. Each sub-directory contains one sample application in two different versions: the `java/` directory will contain the Java-version that can be executed as a pure Java application using XMLVM's iPhone emulator, whereas the `iphone/` directory will contain the ready-to-compile Objective-C source of the same application. The following sample applications are available:

iHelloWorld (Portrait): "Hello World" in portrait mode.

iHelloWorld (Landscape): "Hello World" in landscape mode.

iHelloWorld (Fullscreen): "Hello World" in fullscreen landscape mode.

iFireworks: iPhone version of the fireworks application.

aFireworks: Android version of the fireworks application. The `dist/demo/afireworks/` directory will contain the cross-compiled version for the iPhone. It is possible to import directory `src/demo/afireworks/` as an Android project under Eclipse (requires the Android SDK). In this case aFireworks can be run inside the Android emulator.

Xokoban: Android version of the Xokoban application. The `dist/demo/xokoban/` directory will contain the cross-compiled version for the iPhone. It is possible to import directory `src/demo/xokoban/` as an Android project under Eclipse (requires the Android SDK). In this case Xokoban can be run inside the Android emulator.

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.