Java on the KIPR Link
Markus Klein
Vienna Institute of Technology (TGM)

Java on the KIPR Link

1. Introduction

The purpose of Botball[1] is to motivate students for building and programming autonomous robots. Most new students to Botball do not have any experience with programming. Therefore the way of developing the robot software has to be very beginner friendly and easy to use, but at the same time very powerful.

So what is the best language for Botball? I don't think there is a best choice. Every language has its own advantages and disadvantages. C for example is a very powerful fast and processor-near language so it is quite good for a robot controller doing a lot of basic things such as controlling servos and motors, reading sensor values and so on. But for students who have not yet learned anything about programming or how a computer works, it would be very hard to learn and understand C code. If you want to develop a complex C program, you have to know how pointer arithmetic and memory access works.

Another well-known possibility is Java[2]. Java is much easier to learn and also pretty powerful and reliable. Of course it is not as fast as compiled languages such as C or C++, but in the most cases it will make no real difference. Java is better for beginners, compared to C it doesn't require special knowledge about how a computer works. Java is a higher level language. It provides automatic garbage collection to handle memory management and offers an excellent set of basic data structures such as lists, queues and maps. In C everything has to be built from scratch. The object-oriented paradigm is also very helpful to structure the program to improve the maintainability.

In most schools in Austria students, like me, get started with programming using Java and not C or C++. Therefore I would consider myself a good Java programmer, but I never wrote a bigger project using C.

Further alternatives are event-based frameworks using a scripting language such as node [3] or in general scripting languages, but these can be implemented very easily, because the JVM (Java Virtual Machine)[4] supports a huge amount of additional languages including javascript, scala and clojure.

2. State of the Art

The JamVM[5] which is currently preinstalled on the KIPR Link[6] is not working. I do not know why this version does not work yet. If I try to run a program, the VM outputs "segmentation fault" and stops. Unfortunately there is also no javac java compiler, which is working out-of-the-box. So an alternative Java Runtime Environment (JRE)[7] had to be installed to get the java compiler running.

3. Design Approach

My goal is to provide a Java-Framework which is:

- **reliable:** using a stable Java VM
- **fast:** wrapping the native libkovan for accessing the robot functions
- **easy to learn and to use**
- **object oriented**

The approach is based on wrapping the existing native libkovan[8] with Java using the Java Native Interface[9]. This is basically the same as the CBCJVM[10] which is the Java implementation for the CBCv2[11]. The CBCJVM also wraps over the libcbc. In this approach the actual implementations are written in C and do not have to be implemented newly. See Figure 1 for a detailed overview of the whole framework.

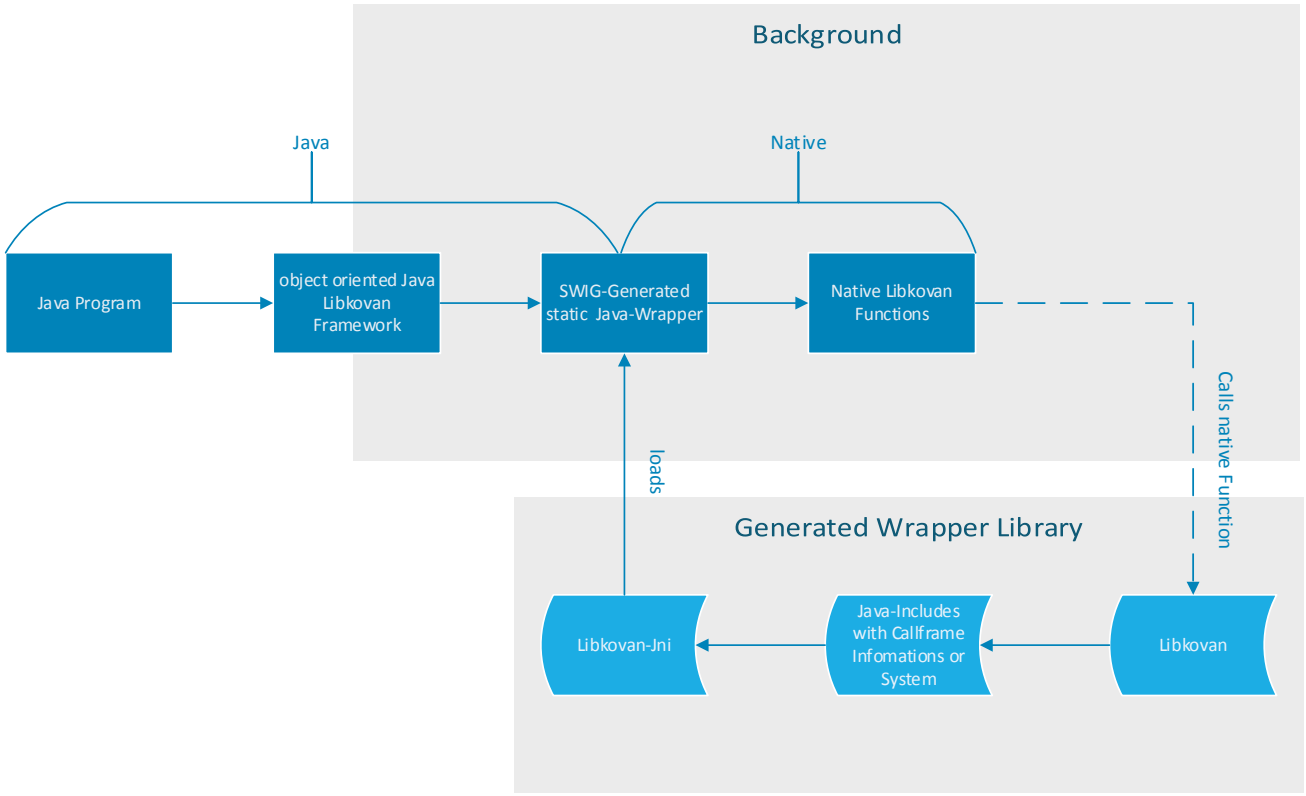

Figure 1 The Approach - framework overview

All parts of the framework are described in detail one by one in the following subsections.

3.1. Java Libkovan

The Java Libkovan provides the object-oriented classes wrapping the functions from the libkovan c standard library. The static java wrapper has been split up into multiple classes to provide a reasonable and clean object oriented usage. The client program only interacts with these classes and all further steps until the final call of the native function from the libkovan library are managed automatically and happen in the background. It is basically an object-oriented wrapper, which wraps the actual libkovan JNI wrapper.

3.2. Static Java Wrapper

The Static Java Wrapper provides the actual wrapper for the libkovan JNI library. This

wrapper is actually only a representation of all functions from the libkovan as static methods. The user should only use the highest lever wrapper (Java Libkovan), since it does not make sense to code java and use the static methods. These functions are only used by the final wrapper itself which was described in the previous subsection. The static java wrapper is auto generated using Simplified Wrapper and Interface Generator (SWIG)[12]. See chapter 4 for more detail about the implementation.

3.3. Libkovan JNI

The Libkovan JNI is the native part of the wrapper. It is a library built out of two files.

3.3.1. Wrapper C File

This file is an automatically generated wrapper file and provides an interface for communication between java and the libkovan C library. It contains all function definitions, the conversion from the java data types to the native data types and includes the system specific jni.h file, which contains the type definitions and information about the system's call stack. The call stack is a special stack where all called functions are stored. See figure 2 below for more details. In addition it offers exception handling for native functions.

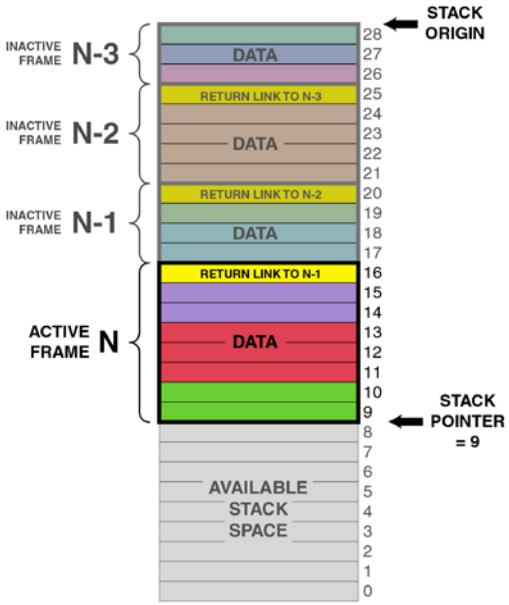

Figure 2 Example of a call stack

3.3.2. Libkovan

This is the pure libkovan library. It is just linked with the java wrapper in the compilation to the final wrapper library.

4. Implementation

4.1. Setting up the Java VM

Before starting with implementing the wrapper library a running JVM (Java Virtual Machine) had to be setup. Currently, there is no running JVM as already mentioned in chapter 2 "State of the Art". I think it would be much easier to set up a new VM than searching the error in the

existing one. The first decision was to use Java Embedded, a special JRE(Java Runtime Environment) which is designed for embedded systems such as ARM, the Power Architecture and x86. It offers the complete Java SE and a JRE with a Just-In-Time(JIT)[13] compiler. The problem is though, that Java Embedded is only free for developing reasons and not for deployment. Fortunately there are a few other open source Java VMs. One of them is the JamVM[5]. The JamVM is a very lightweight VM which needs less memory capacity, supports the ARM Architecture and works with the GNU Classpath[14], an open source implementation of the Java core classes.

4.2. Generating the Wrapper Library

Libkovan contains more than 100 functions, so writing a wrapper by hand would take a lot of time. Therefore, the decision was made to generate the wrapper using SWIG. SWIG is able to generate the C wrapper and the Java wrapper out of an interface file. The interface file looks nearly like a header file and it basically does the same. All function, enum and struct definitions are there.

4.3. Compiling the Wrapper Library

One of the most difficult parts in this project was to compile the wrapper library without a JDK(Java Development Kit). The JDK is needed because system specific header files have to be included for compiling the SWIG[12]-generated libkovan c wrapper which is described in chapter 3.3.1. The JDK is only available for ARM v6/7 processors, but not for ARM v5. As this project does not use the official java core class libraries, it also does not use the official header files. The GNU Classpath provides all required header files. So only GNU Classpath had to be built and installed.

4.3.1. Fixing the Java Compiler

Before we were able to compile the generated java files, we had to get the Java compiler running. This was done really easily, since the main problem was that there is no class path set. This was fixed by simply setting a classpath environment variable or using the “classpath” flag with the path to the java classes.

4.4. Java Object Model

The generated wrapper consists only of one class in which all functions from the libkovan are represented as static methods. In order to provide a well-structured framework the generated Java JNI wrapper has been wrapped one more time. Every class supports one robot component such as cameras, servos or motors. This way every component has its own component class instance and is separated from others.

Everyone familiar with CBCJVM will be able to develop with this Framework. The Syntax is nearly the same. For a short example see Listing 1.

5. Comparison of maintainability with a simple line following program: Java vs. C

In this section I will compare the same very simple line following program once written in C and once in Java.

5.1. C

```
while(true){
	if(analog10(0) < 600){
		motor(0, 100);
		motor(1, 0);
	}
	else if(analog10(1) < 600){
		motor(0, 0);
		motor(1, 100);
	}
	else{
		motor(0, 100);
		motor(1, 100);
	}
}
```

Listing 1: A simple line following program in C

5.2. Java

```
Motor leftMotor = new Motor(0);
Motor rightMotor = new Motor(1);
Sensor leftSensor = new Sensor(0);
Sensor rightSensor = new Sensor(1);

while(true){
	if(leftSersor.getValue() < 600){
		leftMotor.drive(100);
		rightMotor.drive(0);
	}
	else if(rightSensor.getValue() < 600){
		leftMotor.drive(0);
		rightMotor.drive(100);
	}
	else{
		leftMotor.drive(100);
		rightMotor.drive(100);
	}
}
```

Listing 1: A simple line following program in Java

If we take a closer look, we will see that the code is much cleaner and more readable, because we immediately know which motor speed we set or which sensor value we read. Of course we could make the C code clean if we would use constants, but the Java code has a much better maintainability. The real difference in this example is not all that big, but if the code grows up to a few thousand lines, multiple classes will keep the code much more maintainable.

6. Conclusion

Wrapping the native robot library is very lightweight and also pretty fast, because the “real” implementations of the robot functions are written in native c. This concept has already been successfully used with the CBCJVM on the CBCv2. Another great advantage is the maintainability. New features or functions of libkovan can be implemented easily. The aim of

the project, to make it easier for beginners to program the robot and especially for those who have learned Java basics already at school, was successfully shown. In addition, the maintainability and reliability of Botball programs are improved and give everyone the opportunity to develop in one’s favorite programming language.

The next step is to improve the development process. Currently there is no comfortable way to write and debug Java programs for Botball and the KIPR Link. Moreover, an eclipse plugin which automatically uploads the program and is able to debug the program would further improve programming for the KIPR Link in Java.

7. Acknowledgement

First I want to thank Dipl.-Ing. (FH) Mag. Dr.techn. Gottfried Koppensteiner and the whole PRIA staff who made all this possible. In addition great thanks to Wienerberger AG for the financial support.

References

[1] KISS Institute for Practical Robotics: Botball. http://botball.org/, 2013.

[2] Programming Language Java. http://java.com/, November 2005.

[3] Node Javascript. http://nodejs.org/.

[4] Java Virtual Machine. https://en.wikipedia.org/wiki/Jvm, June 2013.

[5] JamVM. http://jamvm.sourceforge.net/, 2010.

[6] KISS Institute for Practical Robotics: Link. http://www.kipr.org/products/link/, 2013.

[7] Java: Runtime Environment. http://www.oracle.com/technetwork/java/javase/downloads/index.html, April 2013.

[8] Braden McDorman: Libkovan C Standard Library. https://github.com/kipr/libkovan, May 2013.

[9] Java: Native Interface. http://docs.oracle.com/javase/7/docs/technotes/guides/jni/.

[10] McDorman B., Woodruff B., Joshi A., Frias J.: "CBCJVM: Applications of the JavaVirtual Machine with Robotics". Presented at GCER 2010.

[11] KISS Institute for Practical Robotics: CBCv2. http://www.kipr.org/products/cbc-robot-controller.

[12] Simplified Wrapper and Interface Generator. http://www.swig.org/, May 2013

[13] Wikipedia: Just In Time Compiler. http://en.wikipedia.org/wiki/Just-in-time_compilation, May 2013.

[14] GNU Classpath: open source Java core class implementation. http://www.gnu.org/software/classpath/, July 2009.