in Proceedings of 2nd Asian Workshop on Aspect-Oriented Software (AOAsia 2006), Tokyo, Japan, September 2006.

Retargetable Compilation with XSLT and Aspects

Andreas Wagner
Computer Science Department, University of Saarland, Germany
awagne@studcs.uni-sb.de

Stephan Diehl
Computer Science Department, University Trier, Germany
diehl@acm.org

Abstract

In this paper we show how XSLT *and aspect-oriented programming can be used to implement retargetable compilers. New devices can be specified using an* XML*-based device specification language. A retargeting generator produces a new compiler from this specification, actually it only produces a device specific kernel which is used by the compiler. Both the retargeting generator and the generated compiler are implemented by* XSLT *scripts that produce* Inject/J *scripts which weave code into a kernel Java program.*

As an example we look at remotely programmable robots. Programs are sent in an XML*-based programming language called* roboXL *over the internet to the server. A compiler on the server translates them into Java programs for the specific robot. The Java program is then transferred to and executed by the robot.*

1. Introduction

Aspect-oriented programming allows to specify code fragments for different properties (concerns, aspects) of a system separately and to compose (weave) them to build a version of the system [6]. Typical aspects are synchronization or access control. In this paper we look at compilers and show how to implement their target platform or target device by aspects.

In our example application we use Java-programmable robots as target devices. Instead of using Java directly to program the robots, programs should be written in an XML-based language called roboXL, such that they can be constructed and manipulated by different XML-based tools. Moreover, the programs can be send over the network and replaced at runtime. Robots typically differ in what sensors and effectors they have and how these can be controlled. Motors for example can change speed, slow down, accelerate, break, and so on. To handle this we allow to extend the XML-Schema of roboXL to create a device specific version of the language.

To build our robots we use *Lego Mindstorms* [7]. By replacing the firmware of the robot by a Java virtual machine [3] programmers are no longer restricted to using a proprietary visual programming language but can actually download Java byte-code into their robot's brain, a special Lego brick called RCX. As new similar robot construction systems like Leonardo or JCX [9] enter the market the need for retargeting the compiler to their APIs and runtime environments arises.

A simple and elegant way to implement compilers is by using XSLT scripts [4]. We solve the retargetability problem by implementing the target device as an aspect. More precisely we implement a retargeting generator as well as the generated compiler with XSLT and AOP only. As long as no complex program analyses are required, the proposed approach is more maintainable and extendable compared to directly implementing the generator with classical compiler techniques in Java [1].

The rest of the paper is organized as follows. Section 2 introduces the language roboXL, then in Section 3 our general approach to retargetable compilation is presented. Section 4 and Section 5 show how compilation and retargeting work in our approach. Finally, related work is discussed in Section 6 and Section 7 concludes the paper.

2. Remote Programming

As a running example we use what is called a *linebot* in the Lego Mindstorm tutorials: a robot that follows a dark line on the ground. The robot has a light sensor and two motors. If both motors rotate in the same direction, it will drive in this direction, if they rotate in opposite directions, the robot turns. The program written in roboXL is shown in Figure 1. The language was designed as a target language

for visual programming tools, as well as to allow transformation, search, and transfer with XML tools.

roboXL provides control structures for sequential, parallel, repeated and conditional execution. The most important elements of roboXL are <SENSOR> terms in expressions to read values and <MOTOR> statements to control motors. In the example program the robot tests in an endless loop (counter is set to zero) whether the value of a light sensor (kind is `light`) measured as a percentage (mode is `pct`) is less than 50%. The rotation direction and speed of the motors is then adjusted as mentioned before. Finally, to provide computation time to other threads on the robot, the command <SLEEP> suspends the current thread for 500 milliseconds.

```
<XL-SEQUENCE>
 <LOOP counter="0">
  <SWITCH>
   <EXPRESSION op="less" type="bool">
    <SENSOR id="1" type="int">
       <KIND>light</KIND>
       <MODE>pct</MODE>
    </SENSOR>
    <VALUE><INT>50</INT></VALUE>
   </EXPRESSION>
   <CASE>
    <VALUE><BOOL>true</BOOL></VALUE>
    <MOTOR id="1">
      <POWER>4</POWER><FWD/>
    </MOTOR>
    <MOTOR id="2">
      <POWER>4</POWER><FWD/>
    </MOTOR>
   </CASE>
   <CASE>
    <VALUE><BOOL>false</BOOL></VALUE>
    <MOTOR id="1">
      <POWER>4</POWER><FWD/>
    </MOTOR>
    <MOTOR id="2">
      <POWER>4</POWER><BWD/>
    </MOTOR>
   </CASE>
  </SWITCH>
  <SLEEP unit="msec">500</SLEEP>
 </LOOP>
</XL-SEQUENCE>
```

Figure 1. Example roboXL program `FollowMe.xml` lets robot follow dark line.

In the rest of the paper we will look at the <SENSOR> tag to illustrate how roboXL can be extended, compiled and retargeted.

The device-independent parts of the language roboXL are defined by an XML Schema which is contained in file `EXOS.xsd`. It defines the control structures and the basic types for motor and sensor tags as shown in Figure 2.

```
<xsd:complexType name="Sensor">
  <xsd:attribute name="id"
     type="SensorId" use="required"/>
  <xsd:attribute name="type" use="required">
    <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="int"/>
          <xsd:enumeration value="float"/>
          <xsd:enumeration value="bool"/>
        </xsd:restriction>
      </xsd:simpleType>
  </xsd:attribute>
</xsd:complexType>
```

Figure 2. Device-independent schema definition of the type `Sensor` in file `EXOS.xsd`.

These types can be refined for a specific robot using inheritance. For example, for the RCX we extend the type `Sensor` by the parameters `KIND` and `MODE` and use the resulting type for the `SENSOR` tag and define the type `SensorId` as the integers from 1 to 3 as shown in Figure 3. In our example, we define three kinds of RCX sensors: raw, touch, temperature and light. For each of these sensors we can either access the measured values as raw (absolute values), boolean or percentage data:

The underlined line in Figure 1 thus is an RCX specific <SENSOR> term and reads the value of the light sensor 1 as an integer. This integer value is interpreted as a percentage.

3. Retargetable Compilation

Apart from its XML-based, illegible syntax roboXL is very similar to Java. There are at least three approaches that we could follow to translate a roboXL program into a Java program for the RCX or another device:

specific compiler Write a compiler that simply generates a RCX specific Java program from the roboXL program. Obviously, this requires to write n compilers for n different target devices.

specific kernel + generic compiler Separate the RCX specific parts from the general parts. Put the specific parts in a kernel and write a compiler that generates a Java program that accesses specific functionality through the kernel. As a result we have to implement only one compiler, but n kernels for n different devices. This approach also has the advantage that it

```
<xsd:element name="SENSOR">
  <xsd:complexType><xsd:complexContent>
      <xsd:extension base="Sensor">
        <xsd:sequence>
          <xsd:element ref="KIND"/>
          <xsd:element ref="MODE"/>
        </xsd:sequence>
      </xsd:extension>
  </xsd:complexContent></xsd:complexType>
</xsd:element>

<xsd:element name="KIND">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="raw"/>
      <xsd:enumeration value="touch"/>
      <xsd:enumeration value="temp"/>
      <xsd:enumeration value="light"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>

<xsd:element name="MODE">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="raw"/>
      <xsd:enumeration value="bool"/>
      <xsd:enumeration value="pct"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>

<xsd:simpleType name="SensorId">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="1"/>
    <xsd:maxInclusive value="3"/>
  </xsd:restriction>
</xsd:simpleType>
```

Figure 3. Device-specific definition of the tags `SENSOR`, `MODE` and `KIND`, and the type `SensorId` in file `RCX.xsd`.

facilitates extending or changing the roboXL language without modifying the device specific parts.

generated kernel and generic compiler Use a compiler as above, but in addition separate generic (reusable) and specific parts of the kernel and write a generator that produces a specific kernel from a device specification. This approach allows us to change the device without having to rewrite the kernel, but instead it is generated from a specification of the new device. So retargeting is reduced to writing n device specifications for n different devices. The generic kernel and compiler have to be written once. Reducing retargeting to writting a device specification not only reduces the amount of work to be done, but also the possible sources of errors.

For the latter approach Figure 4 shows both the compilation and retargeting processes as well as the files involved.

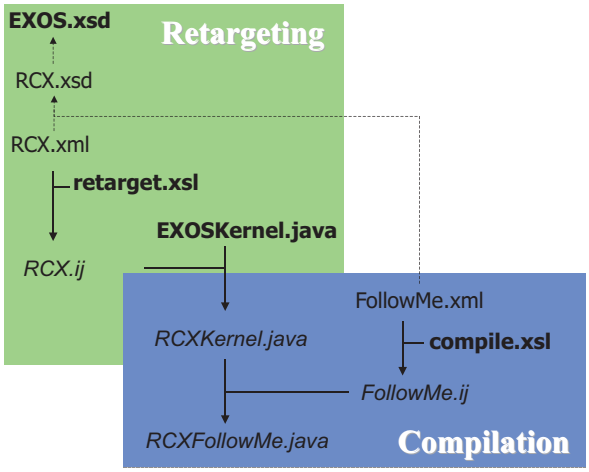

Figure 4. Compilation and Retargeting: Files in bold face are provided by our framework, files in normal font face are user defined and files in italics are generated.

The whole process was implemented as a Java program using the Java API for running the scripts[1] and the resulting Inject/J scripts were executed externally [5]. As a matter of fact, the XSLT scripts can be even run with a XSLT enabled web browser like the one most people use today.

4. Compilation

Basically, compilation works by weaving code into a device specific kernel. First the user writes a roboXL program, which is then transformed by the XSLT script `compile.xsl` into an Inject/J script. This script (see for example Figure 5) then changes the device specific kernel by adding member variables to the kernel, as well as program code to its `main()` method.

The interesting point about `compile.xsl` is that it does not have to know all tags of the source language. In the roboXL extension for the RCX we added parameters to

[1] Actually, we use the JAXP1.2 API, which provides the validating XERCES2 parser instead of the Apache XALAN parser included in Sun's current Java runtime API.

```
add to members ${ String strArg1="1", strArgint="int", strArglight="light",
                   strArgpct="pct", strArgPOWER="POWER", strArg4="4",
                   strArgFWD="FWD", strArgEmpty="", strArg2="2", strArgBWD="BWD"; };

script FollowMe {
 in class 'ExOSKernel' do { foreach method 'main(*)' do {
   before ${ ExOSKernel kern = new ExOSKernel();
             while(true) {
               kern.boolStack.push((((Integer)
                 kern.readSensor(kern.strArg1,kern.strArgint,
                     kern.strArglight,kern.strArgpct )).getValue()<50));
               if (kern.boolStack.top()== true) {
                  kern.controlMotor(kern.strArg1,kern.strArgPOWER,kern.strArg4);
                  kern.controlMotor(kern.strArg1,kern.strArgFWD,kern.strArgEmpty);
                  kern.controlMotor(kern.strArg2,kern.strArgPOWER,kern.strArg4);
                  kern.controlMotor(kern.strArg2,kern.strArgFWD,kern.strArgEmpty); }
               if (kern.boolStack.top() == false) {
                  kern.controlMotor(kern.strArg1, kern.strArgPOWER, kern.strArg4);
                  kern.controlMotor(kern.strArg1, kern.strArgFWD, kern.strArgEmpty);
                  kern.controlMotor(kern.strArg2, kern.strArgPOWER, kern.strArg4);
                  kern.controlMotor(kern.strArg2, kern.strArgBWD, kern.strArgEmpty); }
               kern.boolStack.pop();
               try Thread.sleep(500); catch (InterruptedException ie) {} }
          }$; } } }
```

Figure 5. Inject/J **script** `FollowMe.ij` **generated from the** roboXL **program** `FollowMe.xml` **by the** XSLT **script** `compile.xsl`**.**

the sensor tag. As can be seen in Figure 6, the XSLT script does not know these additional parameters, but it translates the roboXL term `<SENSOR>` into a call to the method `kern.readSensor()`. The generated call is underlined in Figure 5. The script passes the values of the attributes `id` and `type` as well as the values of all tags contained within the `<SENSOR>` tag as parameters to this method. As we will see later, the method `readSensor()` will be generated from the device specification and thus will know how to handle these values. Finally, note how the type attribute is used to cast the return value of the method call.

As reduced Java virtual machines, like the one that runs on the RCX, might not perform garbage collection, our compiler collects all String constants in the program code and defines them once as member variables of the kernel (e.g. `strint`, `strFWD`, ...) to avoid repeated allocation in loops.

5. Retargeting

Basically, retargeting works by weaving code into a generic kernel to produce a device specific one. The generic kernel contains auxiliary methods, state (e.g. an explicit recursion stack), and a scheduler for parallel threads.

To get a specific kernel, we first have to write a specification of the new device. It consists of the Java packages, that have to be imported, variables to maintain state, initialization code and specifications of the arguments of the `readSensor()` and `controlMotor()` methods as well as code to implement these on the specific device. Figure 7 shows parts of the specification for the RCX.

So, the device specification is basically an aspect written in an XML-based language, which can be easily translated by the XSLT script `retarget.xsl` into an Inject/J script. As can be seen in Figure 8, the script `retarget.xsl` actually is a mix of three languages: Java, Inject/J and XSLT

First the XSLT processor interprets the XSLT tags and replaces them by Java code extracted from the device specification, next the instruction `add to members` is interpreted by Inject/J and finally, the resulting Java source code is compiled by a Java compiler.

The resulting Inject/J script for our above example specification is shown in Figure 9.

6. Related Work

roboXL is not the first XML-based robot control language, see for example RoboML [8] which is more complex and puts a focus on data exchange between robots. When designing roboXL we tried to stay close to Java while keep-

```
<xsl:when test="$elem='SENSOR'">
 (( <xsl:choose>
     <xsl:when test="@type='bool'">Boolean</xsl:when>
     <xsl:when test="@type='int'">Integer</xsl:when>
     <xsl:when test="@type='float'">Float</xsl:when>
    </xsl:choose>
  ) kern.readSensor( kern.strArg<xsl:value-of select="@id"/>,
                     kern.strArg<xsl:value-of select="@type"/>
                     <xsl:for-each select="*">
                      , <xsl:choose>
                          <xsl:when test=".=''">kern.strArgEmpty</xsl:when>
                            <xsl:otherwise>
                              kern.strArg<xsl:value-of select="."/>
                            </xsl:otherwise>
                        </xsl:choose>
                    </xsl:for-each> )).getValue()
</xsl:when>
```

Figure 6. Excerpt of the XSLT script `compile.xsl` that translates a roboXL program into an Inject/J script. Here we only show the part that translates the `<SENSOR>` tag.

ing some of the concepts of Lego's visual programming language RoboLab.

XSLT as a tool for program generation is discussed at length in [4]. SmartTools combines AOP with XSLT to generate programming environments[2], while we focus on the retargeting of compilers.

7. Conclusions

To take advantage of XML tools we developed an XML-based language to program robots. The language can be easily translated into Java, but there are robot specific parts that would require re-implementation of such a compiler whenever we choose a new target robot. In order to facilitate retargetability, we separated the compiler and the device specific parts and put the latter into a specification file. From this specification a device specific kernel is generated. For compilation as well as retargeting we use XSLT scripts that produce Inject/J scripts which weave code into kernel modules.

In conclusion, we found that the approach of defining an XML-based application-specific aspect language and then converting it with XSLT to Inject/J scripts or the like is both simple and powerful. It allowed us to concentrate more on separating the reusable and the specific parts and less on the implementation of the conversion process itself, because it was often straightforward having tools like XSLT and Inject/J at hand.

References

[1] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, Cambridge, 1998.

[2] I. Attali, C. Courbis, P. Degenne, A. Fau, J. Fillon, C. Held, D. Parigot, and C. Pasquier. Aspect and XML-oriented Semantic Framework Generator: SmartTools. In *Proceedings of 2nd Workshop on Language Descriptions, Tools and Applications LDTA'02*. Electronic Notes in Computer Science, Elsevier, 2002.

[3] B. Bagnall (webmaster). lejOS – Java for the RCX. Open source project at `http://lejos.sourceforge.net`, 2004.

[4] J. C. Cleaveland. *Program Generators with XML and Java*. Prentice Hall, 2001.

[5] Inject/J Team. Inject/J – Source Code Transformation at your Fingertips. `http://injectj.fzi.de`, 2004.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag LNCS 1241, 1997.

[7] Lego Company. Lego Mindstorms. `http://mindstorms.lego.com`, 2004.

[8] M. Makatchev and S. K. Tso. Human-Robot Interface Using Agents Communicating in an XML-Based Markup Language. In *Proceedings of the 2000 IEEE International Workshop on Robot and Human Interactive Communication RO-MAN2000*, pages 270–275, Osaka, Japan, September 2000.

[9] Systronix Corporation. JCX – Java Control System. `http://jcx.systronix.com`, 2004.

```
<SPECIFICATION>
  <IMPORTS>josx.platform.rcx.Motor, ...</IMPORTS>
  <VARIABLES>
    private String strlight="light"; private String strpct="pct";
    private Boolean objReturnBoolean=new Boolean(false);
    private Integer objReturnInteger=new Integer(0); ...
  </VARIABLES>
  <INIT> sensors=Sensor.SENSORS; </INIT>
  <READSENSOR> <ARGS>String sensor,String type,String kind,String mode</ARGS>
    <CODE>
      int intKind=0; int intMode=0;
      if (kind.equals(strlight)) intKind=SensorConstants.SENSOR_TYPE_LIGHT; ...
      if (mode.equals(strpct)) intMode=SensorConstants.SENSOR_MODE_PCT; ...
      int sensorNr=stringToInt(sensor)-1;
      ((Sensor) sensors[sensorNr]).setTypeAndMode(intKind,intMode);
      ((Sensor) sensors[sensorNr]).activate();
      if (type.equals(strint))
          { objReturnInteger.setValue(((Sensor) sensors[sensorNr]).readValue());
            return objReturnInteger; }
      else if (type.equals(strbool))
          { objReturnBoolean.setValue(((Sensor) sensors[sensorNr]).readValue());
            return objReturnBoolean; }
      else { objReturnInteger.setValue(0); return objReturnInteger; }
    </CODE> </READSENSOR> ... </SPECIFICATION>
```

Figure 7. The device specification `RCX.xml` for the Lego robot.

```
add to members ${ public synchronized Object readSensor
 (<xsl:value-of select="/exos:EXOS/exos:SPECIFICATION/exos:READSENSOR/exos:ARGS"/>)
  { <xsl:value-of select="/exos:EXOS/exos:SPECIFICATION/exos:READSENSOR/exos:CODE"/> } }$;
```

Figure 8. Excerpt of the XSLT script `retarget.xsl`. Here we only show the part that generates the device-specific `readSensor()` method.

```
script RCX { in class 'ExOSKernel' do {
    add to imports ${josx.platform.rcx.Motor,...}$;
    in method 'ExOSKernel()' do { before ${ sensors=Sensor.SENSORS; ... }$; }
    add to members ${ private String strlight="light"; private String strpct="pct";
                      private Boolean objReturnBoolean=new Boolean(false);
                      private Integer objReturnInteger=new Integer(0); ... }$;
    add to members ${ public synchronized Object
        readSensor(String sensor, String type, String kind, String mode) {
          int intKind = 0; int intMode = 0;
          if (kind.equals(strlight)) intKind = SensorConstants.SENSOR_TYPE_LIGHT; ...
          if (mode.equals(strpct)) intMode = SensorConstants.SENSOR_MODE_PCT; ...
          int sensorNr=stringToInt(sensor)-1;
          ((Sensor) sensors[sensorNr]).setTypeAndMode(intKind, intMode);
          ((Sensor) sensors[sensorNr]).activate();
          if (type.equals(strint)) {
            objReturnInteger.setValue(((Sensor) sensors[sensorNr]).readValue());
            return objReturnInteger; }
          else if (type.equals(strbool)) {
            objReturnBoolean.setValue(((Sensor) sensors[sensorNr]).readValue());
            return objReturnBoolean; }
          else { objReturnInteger.setValue(0); return objReturnInteger; }  } }$; } }
```

Figure 9. Inject/J **script `RCX.ij` generated from the specification `RCX.xml` by the** XSLT **script `retarget.xsl`.**