An Extensible Approach to Extracting Specification from Device Driver Source Codes Using XSL*

Lei Xiao, Jianfeng Cui, Xiaozhu Xie, Lijuan Liu

School of Computer & Information Engineering, Xiamen University of Technology, Xiamen 361024, China

{lxiao & jfcui & 2011110703 & ljliu}@xmut.edu.cn

Abstract - Generally, device drivers have various implementations depending on the environments such as target devices and operating systems. The concept of device driver specification has been introduced to increase understand ability of device drivers and their implementation. This paper presents an XSL-based tool for extracting a device driver specification from device driver source code. We focus on the extensibility of specification extractions so that any changes to the structures of source code and driver specification can be accommodated without modifications to the tool. In this paper, the architecture of the tool is described and the result of its application to extract device driver specification on Linux platform is provided.

Index Terms - Device drive, XSL, specification extraction.

I. Introduction

Most of today's home appliances and mobile phones use embedded system, which, in many cases, has taken over what mechanical and dedicated electronic systems used to do [1]. Device drivers are essential components of operating system kernel for interfacing software applications with hardware devices. As part of complex operating system, device drivers are considered extremely difficult to develop.

Specification of device driver source codes is an approach to supporting the development of device drivers [2]. The specification is a high-level description on the essential device driver information at the level of design [3]. The reason for device driver specification is that a device engineer spends a lot of time on searching information for related program understanding and maintenance tasks [4]. An abstraction of source code can help developers comprehend software by uncovering relationships between classes, modules, units, functions, etc. Specification extractor is an important tool for reverse reengineering, maintenance of software systems.

Focusing on the extensibility, we have implemented a tool for an extraction of device driver specification. The tool, named XDDSE (eXtensible Device Driver Specification Extractor), supports the evolution. That is, it supports various changes in the structures of source code and specification, and their mapping without any modification to the tool. To support the extensibility, we adopted XSLT (eXtensible Stylesheet Language Transformation) technologies. This paper presents XDDSE and shows its usefulness by applying it for extracting Linux device driver specification.

II . Extensible Device Driver Specification Extractor

Fig.1shows the architecture of XDDSE. XDDSE consists of two components; C modeling language (CML) Generator and Device Driver Specification (DDS) Generator.

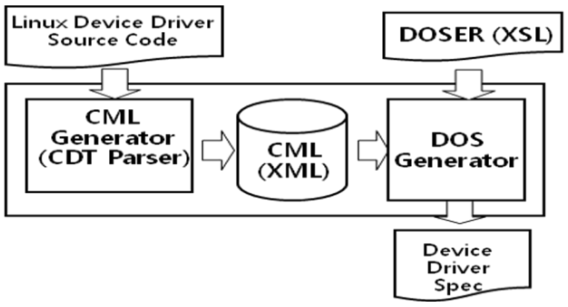

Fig. 1 Architecture of XDDSE

CML Generator extracts semantic information from device driver C source code and generates CML as an intermediate file. CML Generator uses CDT Parser of eclipse plug-in. CML is an XML scheme of C. Using CML, XDDSE extracts device driver specifications from device driver source codes.

DDS Generator generates a device driver specification from CML using DDSER. DDSER (Device Driver Specification Extraction Rule) include code patterns and device driver specification information for extracting device driver specification from device driver C source code.

A. C Modeling Language

C Modeling Language (CML) represents a device driver source code as an XML file. The names of CML node element follow the AST class names of eclipse CDT [5]. For example, TABLE 1 shows a example of CML representation. The second column is the XML representation of the C source code in the first column.

B . Device Driver Specification Extraction Rule

Device Driver Specification Extraction Rule (DDSER) defines rules for extracting specification based on code

* Supported by the Natural Science Foundation of Fujian Province of China under Grant No.2011J05160 to Jianfeng Cui; the funding (type B) from the Fujian Education Department under Grant No.JB12186 to Xiaozhu Xie

patterns in device driver source codes. DDSER consists of five device driver specification elements based on the same structure of the device driver specification; basic, device, bus, file-operations, and advanced. Fig. 2 shows DDSER structure. Each extraction rule file follows W3C document transformation standard XSLT. DDSER defines each element of device driver specification as a module, so that DDSER can be easily modified and extended to support changes in device drivers.

TABLE 1 An Example of CML

#include <stdio.h>
 	
void main(){
printf("Hello!\n");
return;
}
	<C>
	<include>**stdio.h**</Include>
	<FunctionDefinition>
	<DeclSpecifier>**int**</DeclSpecifier>
	<FunctionDeclarator>
	<Name>**main**</Name>
	</FunctionDeclarator>
	<CompoundStatement>
	<FunctionCallExpression>
	<Name>**printf**</Name>
	<LiteralExpression>**"Hello!\n"**
	</LiteralExpression>
	</FunctionCallExpression>
	<ReturnStatement />
	</CompoundStatement>
	</FunctionDefinition>
	</C>

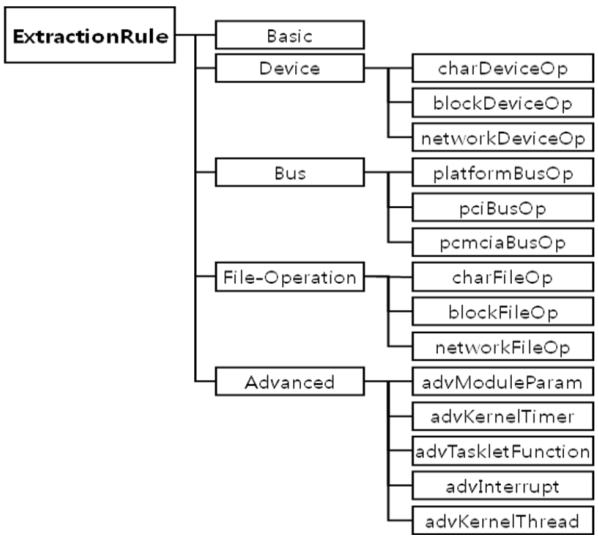

Fig. 2 DDSER Structure

ExtractionRule.xsl is the master extraction rule XSL file of DDSER. It contains information about device driver specification XML representation and rules for determining which module will be used to extract specification elements from a device driver source code.

TABLE 2 shows the modules in ExtractionRule.xsl. The first column represents classification of each module and second column shows its detailed implementation using XSL. Include module shows included XSL files in Extraction Rule.xsl.

TABLE 2 ExtractionRule.xsl

Include	<xsl:include href="platformBusOp.xsl"/> <xsl:include href="pciBusOp.xsl"/> <xsl:include href="charDeviceOp.xsl"/> <xsl:include href="charFileOp.xsl"/> <xsl:include href="advModuleParam.xsl"/> <xsl:include href="advKernelTimer.xsl"/> <xsl:include href="advTaskletFunction.xsl"/> <xsl:include href="advInterrupt.xsl"/> <xsl:include href="advKernelThread.xsl"/>
Basic	<basic> <cpu> arm </cpu> <version>2.6.15.7-gcov</version> <xsl:if test="C/SimpleDeclaration /FunctionDeclaration"> <xsl:apply-templates select= "C/SimpleDeclaration/FunctionDeclaration"/> </xsl:if> </basic>
Device	<device> <xsl:if test="C/FunctionDefinition"> <xsl:apply-templates select= "C/FunctionDefinition" mode="**Device**"/> </xsl:if> </device>
Bus	<bus> <xsl:if test="C/FunctionDefinition"> <xsl:apply-templates select="C/FunctionDefinition" mode="**Bus_operations**"/> </xsl:if> </bus>
File-operations	<file-operations> <xsl:if test="C/FunctionDefinition"> <xsl:apply-templates select="C/FunctionDefinition" mode="**File_operations**"/> </xsl:if> </file-operations>
Advanced	<advanced> <xsl:if test="C/SimpleDeclaration"> <module-param><xsl:apply-templates select="C/SimpleDeclaration" mode="**advanced_module_param**"/> </module-param> </xsl:if> <xsl:if test="C/Macro"> <xsl:apply-templates select="C/Macro" mode="**advanced_kernel_timer**"/> </xsl:if> <xsl:if test= "C/FunctionDefinition/FunctionDeclarator"> <xsl:apply-templates select= "C/FunctionDefinition/FunctionDeclarator" mode="**advanced_thread**"/> </xsl:if> <xsl:if test= "C/FunctionDefinition/FunctionDeclarator"> <xsl:apply-templates select= "C/FunctionDefinition/FunctionDeclarator " mode="**advanced_taklet**"/> </xsl:if> <xsl:if test="C/FunctionDefinition"> <xsl:apply-templates select="C/FunctionDefinition" mode="**advanced_workqueue**"/> </xsl:if> <xsl:if test="C/FunctionDefinition"> <xsl:apply-templates select="C/FunctionDefinition" mode="a**dvanced_interrupt**"/> </xsl:if> </advanced>

In XSL each module is regarded as a function. So, each module and sub-module of DDSER is considered a function. Basic module shows its extraction location of a device driver source code. Device, Bus, and File-operations modules show their sub modules Device, Bus_operations, and File_operations. Advanced module consists of several sub-modules.

TABLE 3 charFileOp.xsl

ExtractionRule.xsl: Caller part

```
<xsl:template
   match="FunctionDefinition"  mode="File_operations">
  <xsl:choose>
    <xsl:when test="contains(child::
     CompoundStatement, 'register_chrdev')">
     <xsl:apply-templates select="ancestor::C/SimpleDeclaration"
      mode="CharPrefix"/>
      <xsl:call-template  name="CharFunctionName"/>
      </xsl:when>
   </xsl:choose>
</xsl:template>
```

charFileOp.xsl

```
<xsl:template  match="SimpleDeclaration"  mode="CharPrefix">
   <xsl:choose>
      <xsl:when test="contains
        (child::DeclSpecifier, 'file_operations')">
        <prefix>
          <xsl:apply-templates select=
            "descendant::ICASTDesignatedInitializer
              /InitializerExpression"  mode="PrefixDetail"/>
        </prefix>
      </xsl:when>
   </xsl:choose>
</xsl:template>
```

```
<xsl:template  match="InitializerExpression"
     mode="PrefixDetail">
   <xsl:if test="not(contains(.,'THIS'))">
      <xsl:if test="contains(ancestor::
         SimpleDeclaration, 'file_operations')">
         <xsl:if test="position() = 2">
            <xsl:value-of  select="substring-before(.,'_')"/>
         </xsl:if>
      </xsl:if>
   </xsl:if>
</xsl:template>
```

```
<xsl:template name="CharFunctionName">
   <xsl:for-each select="ancestor::C/SimpleDeclaration/Declarator
         /ICASTDesignatedInitializer/Name">
      <xsl:if test="not(contains(.,'owner'))">
        <xsl:if test="contains(ancestor::
         SimpleDeclaration, 'file_operations')">
         <xsl:element name="function">
           <xsl:attribute name="name">
             <xsl:value-of select="."/>
           </xsl:attribute>
         </xsl:element>
        </xsl:if>
      </xsl:if>
   </xsl:for-each>
</xsl:template>
```

TABLE 3 represents charFileOp.xsl file. The charFileOp.xsl is the module for character device file operation extraction. The first row represents File-operations module calling part. The ExtractionRule calls the module CharPrefix that extracts prefix of character file-operations. The other rows show an extracting module of the CharPrefix. The CharPrefix module calls its sub-module PrefixDetail, and extracts its detailed specifications.

A device uses a bus (e.g. PCI, USB, and platform). Device driver specifications should provide a bus structure and operations information for bus communication, and Bus module should extract a bus structure and operations specification. Because the bus structure and operations are different depending on the bus type, their specifications are classified by its bus type. Advanced module consists of a set of independent modules. DDSER provides six Advanced modules: module-parameter, kernel-timer, kernel-thread, interrupt, tasklet and workqueue.

TABLE 4 provides an example of advanced module. The first row shows the caller part in ExtractionRule.xsl. The ExtractionRule.xsl calls advanced_module_param module and advMouduleParam.xsl extracts specification from device driver source code.

TABLE 4 advModuleParam.xsl

ExtractionRule.xsl: Caller part

```
<xsl:if test="C/SimpleDeclaration">
   <module-param>
     <xsl:apply-templates  select="C/SimpleDeclaration"
          mode="advanced_module_param"/>
   </module-param>
</xsl:if>
```

advModuleParam.xsl

```
<xsl:template match="SimpleDeclaration"
    mode="advanced_module_param">
  <xsl:for-each select="FunctionDeclarator">
    <xsl:if test=" contains(child::Name, 'module_param')">
      <xsl:element name="param">
        <xsl:for-each select="child::ParameterDeclaration">
          <xsl:if test="position()=1">
            <xsl:attribute name="name"> <xsl:value-of select="."/>
            </xsl:attribute>
          </xsl:if>
          <xsl:if test="position()=2">
            <xsl:attribute name="type"> <xsl:value-of select="."/>
            </xsl:attribute>
          </xsl:if>
           <xsl:if test="position()=3">
            <xsl:attribute name="perm"> <xsl:value-of select="."/>
            </xsl:attribute>
           </xsl:if>
        </xsl:for-each>
      </xsl:element>
    </xsl:if>
  </xsl:for-each>
</xsl:template>
```

III . A Case Study

We apply XDDSE to several Linux device driver source codes. XDDSE supports three device types, including character, block and network devices, and three bus-types, including platform, PCI and PCMCIA. TABLE 5 represents the supported devices, bus types and advanced features. We have applied XDDSE to a simple device source code of character device type, nobus bus type, and several advanced features.

TABLE 5 Supported Linux Device Driver Types

Device Type	**Character**, Block, Network
Bus Type	Platform, PCI, PCMCIA, **Nobus**
Advanced Feature	**Module-parameter**, **Kernel-timer**, **Tasklet-function**, **Interrupt**, Kernel-thread, **Workqueue**

In this section, we give a case study. Several changes can be considered: change of kernel version, appending new device, and change of device driver specification. We can classify those changes into two categories: changes of code pattern and addition of new device driver specification elements.

Changes of code pattern. Changes of code pattern occur when the kernel is changed or new device is appended. Device driver specification is changed and device driver specification requires new information, then extraction code pattern needs to be changed. TABLE 6 represents an old pattern and its modified pattern. To extract name of a device, in the case of old pattern its location is the macro declaration KDSE_cdev, but in modified pattern the device name location is changed to function declaration KDSE_init.

TABLE 6 Changes of Code Pattern Configuration

Element	Old Pattern	Modified Pattern
Device / Name	static struct cdev KDSE_cdev;	static int KDSE_init(void) {…}

TABLE 7 gives an example of change of code pattern in XSL files. In the first row, old code pattern is '_cdev' in variable declaration but in the second row, modified code pattern is changed to '_init' in function declaration. In the case of the pattern '_cdev', the code pattern can only be applied to character device. The modified pattern, however, '_init' is the initializing function of a device driver, so the modified pattern can be applied to various types of devices. Simply modifying DDSER, that is, changes of code pattern can be easily supported without changes in XDDSE tool.

TABLE 7 Result of Changes of Code Pattern

```
<xsl:template match="Macro" mode="CharDev">
….
<xsl:when test="contains(child::Name,'_cdev')">
  <name> <xsl:value-of select="substring-before(.,'_cdev')"/>
  </name>
  ….
</xsl:when>
….
</xsl:template>
```

```
<xsl:template match= "FunctionDefinition"  mode="Device">
<xsl:if test="contains(child::FunctionDeclarator,'_init')">
  <name> <xsl:value-of select="substring-before
     (child::FunctionDeclarator,'_init')"/>
  </name>
  ….
</xsl:if>
….
</xsl:template>
```

Addition of new device driver specification elements. New device driver specification elements need to be added when a new device is added to Linux system. Let us consider a situation where new element of device driver specification is appended. TABLE 8 shows old element and appended element.

TABLE 8 Addition of New Specification Elements

Element	Old Element	Appended Element
Advanced /module-param	None	module-parameter: - name: param1 - name: param2 - name: param3 - name: param4 - name: param5

IV . Related Works

Source code generation and tools are studied in progress actively. For example, WinDriver [7], DriverStudio[8], Driver Development Kit [9] includes source code generation tools. WinDriver offers functions for hardware and kernel information extraction that is needed for driver frame code generation. DriverStudio offers debugging, testing and analyzing software performance tools. Windows Driver Development Kit provides a build environment, tools, driver samples, and documentation to support driver development.

V . Conclusion and Future Works

In this paper, we present a tool XDDSE for extracting device driver specification. XDDSE is designed to support its extensibility by using XSL. A case study is given for illustrating the extensibility of XDDSE.

At present, DDSE mainly covers several kinds of device and bus types. There are, however, other possible classification such as sound and video. Each usage-domain device driver has its own characteristics. They will be included in the device driver specification and the DDSER. In addition, we plan to develop source code generation tool for device drivers based on the same technology, XSL.

Reference

[1] Lee. E. A. "What's ahead for embedded software?," Computer, Volume 33, Issue 9, Sep 2000, pp. 18-26

[2] Mattias O'Nils, Johnny Oberg, Axel Jantsch, "Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces," EUROMICRO'98, 1998, p. 10055

[3] Chikofsky.E.J, Cross.J.H "Reverse engineering and design recovery: a taxonomy," Software, IEEE, Volume 7, Issue 1, Jan 1990, pp. 13-17

[4] Laurent Reveillere, Gilles Muller, "Improving Driver Robustness : an Evaluation of the Devil Approach," Proceedings of the 2001 International Conference on Dependable Systems and Networks, 2001

[5] CDT Parser plug-in, http://www.eclipse.org/cdt/

[6] Yong Hoon Choi, Woo Il Kown, Heung Nam Kim, "Code generation for Linux device driver," 2006. ICACT 2006, pp. 4

[7] WinDriver(http://www.jungo.com/windriver_usb_pci_driver_development_software.html

[8] DriverStudio(compuware) http://www.compuware.co.kr/products/driverstudio/ds/

[9] Windows Driver Development Kit http://www.microsoft.com/whdc/devtools/ddk