Domain Analysis of Device Drivers Using Code Clone Detection Method

Yu-Seung Ma and Duk-Kyun Woo

Domain analysis is the process of analyzing related software systems in a domain to find their common and variable parts. In the case of device drivers, they are highly suitable for domain analysis because device drivers of the same domain are implemented similarly for each device and each system that they support. Considering this characteristic, this paper introduces a new approach to the domain analysis of device drivers. Our method uses a code clone detection technique to extract similarity among device drivers of the same domain. To examine the applicability of our method, we investigated whole device drivers of a Linux source. Results showed that many reusable similar codes can be discerned by the code clone detection method. We also investigated if our method is applicable to other kernel sources. However, the results show that the code clone detection method is not useful for the domain analysis of all kernel sources. That is, the applicability of the code clone detection method to domain analysis is a peculiar feature of device drivers.

Keywords: Device drivers, code clone detection.

I. Introduction

A device driver is a software component which provides an interface between the operating system and specific hardware devices, such as terminals, disks, and network media. However, as device drivers are critical and low-level system codes, they are difficult to implement. Also, they have been noted as a major source of system faults. To overcome these problems, a few studies [1]-[3] have been conducted to verify or test device drivers, and other studies [4]-[6] have been conducted to develop reliable device drivers. The studies mainly try to generate device driver sources using high-level languages, such as specification languages. However, currently, there is no standard or de facto standard specification language for device drivers. To develop a specification language appropriate to device drivers, their domain analysis is fundamental.

Domain analysis [7] is the process of identifying, collecting, organizing, and representing the relevant information in a domain, based upon the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within a domain. It focuses on supporting systematic reuse by capturing both the commonalities and the variations of systems within a domain to improve the efficiency of development and maintenance of those systems. However, domain analysis is very time-consuming and difficult. In the case of device drivers, the difficulty becomes worse because the analysis of device drivers requires deep knowledge of both the system and the device; therefore, systematic and efficient methods for domain analysis of device drivers are required.

The process of domain analysis usually involves at least two steps: a passive step (identifying reusable entities) and an active one (structuring and organizing information) [8]. However, the

Manuscript received Aug. 10, 2007; revised Nov. 23, 2007.

This work was supported by the IT R&D program of MKE/IITA, Rep. of Korea (2008-S-023-01, Development of NanoQplus-Based Sensor Network Simulator).

Yu-Seung Ma (phone: +82 42 860 6551, email: ysma@etri.re.kr) and Duk-Kyun Woo (email: dkwu@etri.re.kr) are with S/W & Content Research Laboratory, ETRI, Daejeon, Rep. of Korea.

previous studies on domain analysis of device drivers [4]-[6] have been only concerned with the active step, and let the passive step be conducted manually. To provide automatic support of the passive step, this paper introduces a new approach which can help the passive step of a domain analysis of device drivers. Our solution uses a code clone detection technique to analyze the similarities of device drivers within the same domain.

Code clones [9], [10] are code portions in source files which are identical or similar to each another. They are introduced for various reasons [10], the most famous of which is the re-use of code by copy-and-paste. Clones are usually considered to be undesirable because they often introduce errors. However, there are several situations in which code duplication seems to be a reasonable or even beneficial design option [11], [12]. Our approach is to make use of this positive aspect of code clones. That is, we seek to validate that code clones are inevitable and helpful to the domain analysis of device drivers.

In contrast to normal programs, which provide diverse functions over diverse domains, device drivers implement specific functions over specific domains. As a result, if device drivers are of the same kind, their behavior is almost the same. In fact, many developers implement device drivers referring to (mimicking) existing device drivers of the same domain. Thus, it seems likely that there would be many pairs of code clones among device drivers of the same domain.

In this paper, we categorize code clones into two groups: intra code clone (intra-cc) and inter code clone (inter-cc). *Intra code clone* means a code clone whose pair exists in the same source file. On the other hand, an *inter code clone* is a code clone of which the two matching parts are in different source files. To validate the above-mentioned expectation, we analyze inter code clones of source files of device drivers using a code clone detection system, CCFinder [10]. This paper only focuses on the inter code clone because we are interested in extracting similarities among different device drivers of the same domain, and we anticipate that different device drivers are implemented as different files.

Another purpose of this paper is to investigate if our method can be useful to other kernel sources besides device drivers. For the purpose, we also analyze inter code clones of other kernel sources.

The remainder of this paper is organized as follows. Section II briefly introduces the code clone detection system, CCFinder, and some of its metrics. Sections III and IV analyze inter code clones of device driver sources and Linux kernel codes, respectively. Section V gives a simple case study, and section VI discusses related works. Finally, section VII concludes the paper.

II. Background

There are many code-clone detection tools such as CCFinder [10], CloneDR [9], and Dup [13]. In our study, we used CCFinder [10] because it has good code detection ability and, above all, it provides metrics related to inter-cc pairs.

CCFinder detects clones with transformation rules and a token-based comparison. Currently, it can detect code clones from source files written in Java, C, C++, COBOL, VB, and C#. This section briefly describes some definitions and metrics that CCFinder uses. CCFinder [10] defines a *clone relation* as an equivalence relation (that is, reflexive, transitive, and symmetric relation) between code portions. A clone relation holds between two code portions if and only if they are the same sequences. A pair of code portions is called a *clone pair* if a clone relation holds between the portions.

CCFinder allows the detection of code clones with four options: minimum clone length, minimum TKS, shaper level, and P-match application. The 'minimum clone length' option defines the number of minimum number of tokens required for a code to be a clone. For example, suppose a file has the following 12 tokens:

a b c x y 1 a b c 2 x y

Assume the value of the minimum clone length is 3. Then, only the portion "a b c" can be a code clone. The portions, "x y," "a b," and "b c," cannot be code clones because their token length is 2.

The minimum TKS define the size of a set of tokens of a code fragment of a code clone. The shaper level option is used to recognize block structure. CCFinder supports four shaper levels: hard shaper, soft shaper, easy shaper, and without shaper. With hard shaper, only a token sequence enclosed by a block is regarded as a clone candidate. With soft shaper, a token sequence which is not split by an outer block boundary is regarded a clone candidate. With easy shaper, an arbitrary token sequence is regarded as a clone candidate, but its length is measured including its un-split token sequence. Without shapers, the boundaries of blocks are neglected, that is, any arbitrary token sequence is a candidate of clone.

The P-match application option is related to variables or function names. Without P-match, the preprocessor replaces all variables or function names with a special token, so that the difference between names is neglected. Both "return x + y" and "return a + a" are transformed into "return $ + $" (here $ is the special token for an identifier), so they are identified as a clone pair. However, with P-match, "return x + y" and "return a + a" are transformed into "return $1 + $2" and "return $1 + $1," respectively, so they are not identified as a clone pair.

In this paper, we use the default setting (minimum clone length=50, minimum TKS=12, shaper level=2-soft shaper, P-match application=use) provided by CCFinder.

CCFinder provides several metrics related to code clones,

which are calculated against a code clone or a file. The following two metrics that are used in this paper are calculated against a file and related with an inter code clone:

- **NBR(*f*):** the number of source files that include one or more code fragment of the inter code clones related to the file *f*
- **RSA(*f*):** the ratio (percentage) of tokens of the file *f* that are covered by inter code clones

The NBR value is an integer whose value is more than zero, and a large NBR value means that there are many similar files. An RSA value represents a ratio of similarity between the files. If a file has an RSA value close to 100%, then it is possible that the file was created by copying other files.

III. Inter Code Clones of Device Drivers

This section shows how many inter code clones exist among source codes of device drivers. For the study, we utilized Linux because its source code is freely available; moreover, it contains sources of diverse device drivers. The source of Linux consists of kernels and drivers written in C. Considering the top-level directory of the Linux source as LINUX_SRC, the sources of device drivers are mainly under the LINUX_SRC/drivers directory and arranged in appropriate lower directories according to their function or bus types. For example, sources of USB network device drivers and USB storage device drivers are located under the LINUX_SRC/drivers/usb/net directory and the LINUX_SRC/drivers/usb/storage directory, respectively. Furthermore, some directories contain sources of various device drivers of the same domain. For example, the LINUX_SRC/drivers/usb/storage directory contains sources of storage device drivers for diverse vendors such as Sony, Datafab, and Samsung.

Table 1. Inter code clones among the device driver sources of the LINUX_SRC/drivers directory.

Location	Number of C files	Files having inter-cc		
		Number	Average NBR	Average RSA
/acpi	152	38 (25%)	3.66	0.12
/char	267	160 (60%)	12.17	0.22
/infiniband	110	54 (49%)	2.02	0.10
/input	105	73 (70%)	7.53	0.24
/isdn	163	105 (64%)	7.90	0.26
/media	322	235 (73%)	6.63	0.22
/mtd	137	70 (51%)	1.90	0.24
/net	475	336 (71%)	6.90	0.16
/scsi	232	153 (66%)	3.56	0.16
/usb	205	146 (71%)	8.44	0.21
/video	181	119 (66%)	4.52	0.17
total	2,349	1,489 (63%)	6.65	0.19

The analysis uses the Linux source of version 2.6.10. In the source, the LINUX_SRC/drivers directory consists of 63 sub-directories. Because our intention is to compare various driver sources of the same domain, we chose the sub-directories that consist of more than 100 driver sources except header files. Then, a total of 11 sub-directories were selected as shown in Table 1.

Table 1 shows summarized information related to inter code clones of the device driver sources under the 11 sub-directories, which was analyzed with CCFinder. The location column represents the relative paths of the directories from the LINUX_SRC/drivers directory. The second column shows the number of C source files. The number of files is calculated by including files of nesting sub-directories (sub-directories of a sub-directory). Information about the device driver sources that contain inter code clones are on the right side of Table 1. Among the total of 2,349 driver sources, about 63% of the sources contain at least one inter code clone. The average NBR value is 6.65. Based on this value, we may consider that there are, on average, six to seven similar device driver sources in the same domain.

Next, we investigate the degree of similarity between the driver sources by examining the RSA values. The average RSA value is 0.19, which means that about 19% of the codes are the same or similar among different driver sources that have inter code clones. Contrary to our expectation, the RSA value appears to be small. However, this can be partly explained by the fact that the number of files in each directory does not indicate the number of device drivers in the same domain. For example, some device drivers consist of files from two or more sources. Assuming that there are four files, $f1$, $f2$, $f3$, and $f4$, in a specific directory, it is possible that $f1$ and $f2$ implement one device driver, and $f2$, $f3$, and $f4$ implement another device driver. In this case, $f2$ can be considered a common library module. That is, the directory which consists of four files implement only two device drivers in the example.

Figure 1 shows the distribution of individual RSA values of the 2,349 driver sources shown in Table 1. In the figure, we can see files whose RSA values are above 0.9, which mean they are almost identical to other files. Although most of the RSA values are concentrated in values less than 0.1, many files have RAS values above 0.5, which demonstrates that they are similar.

To give more detailed information, we analyze inter code clones of the LINUX_SRC/drivers/input directory by examining its sub-directories. We chose this directory because the sources of input device drivers are easy to present and

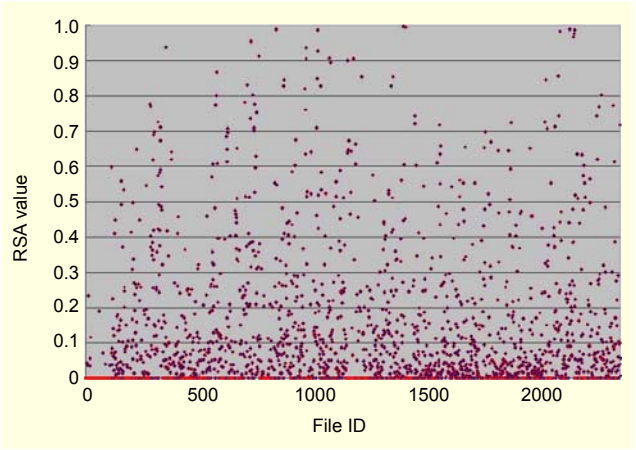

Fig. 1. Distribution of the RSA values of the device driver sources in the LINUX SRC/drivers directory.

Table 2. Inter code clones among the device driver sources of the LINUX_SRC/drivers/input directory.

Location	Number of C files	Files having inter-cc		
		Number	Average NBR	Average RSA
/input/gameport	5	3 (60%)	1.33	0.26
/input/joystick	27	22 (81%)	10.59	0.20
/input/keyboard	14	11 (79%)	7.27	0.36
/input/misc	7	7 (100%)	4.71	0.22
/input/mouse	14	10 (71%)	3.70	0.15
/input/serio	17	4 (24%)	2.00	0.15
/input/touchscreen	12	10 (83%)	12.90	0.39
/input	9	6 (67%)	4.33	0.17
total	105	73 (70%)	7.53	0.24

understand. Table 2 gives detailed information about inter code clones of the LINUX_SRC/drivers/input directory.

The LINUX_SRC/drivers/input directory contains seven sub-directories and nine files, which are located there directly. The RSA value of the directory was 0.24 on average. This means that about 24% of the codes are similar with other files on average. Investigating the RSA values for each individual file in the directory, the lowest RSA value was 0 and the highest RSA value was 0.87. In fact, in the case of the files whose RSA values are over 0.5, their codes seem to be very strongly similar. Table 4 compares part of the sources, gunze.c and mtouch.c, whose RSA values are 0.56 and 0.61, respectively. We highlighted the codes in Table 4 that are different according to their device dependent features. Surprisingly, the codes are very similar and differences are mainly due to device dependent features, such as name and abs values. This suggests that it is possible to generate device driver sources using a template code where some values are constituent.

Table 3. Inter code clones among the device driver sources of the LINUX_SRC/kernel directory.

Location	Number of files	Files having inter-cc		
		Number	NBR	RSA
/kernel/irq	9	0 (0%)	0	0
/kernel/power	11	0 (0%)	0	0
/kernel/time	3	0 (0%)	0	0
/kernel	77	7 (9%)	1.28	0.07
total	100	7 (7%)	1.28	0.07

Although we only give detailed information for input device drivers, which have relatively high RAS values, other kinds of device drivers show similar patterns.

IV. Inter Code Clone of Kernel Source

Although device drivers of the same domain have lots of inter code clones, this characteristic may not be peculiar only to device drivers. Kernel sources also may have many inter code clones among them. To investigate this issue, we examine code clones of kernel sources and compare them with those of device drivers. For this purpose, we investigated the LINUX_SRC/kernel directory using the same procedure described in the previous section. Table 3 summarizes code clones of the LINUX_SRC/kernel directory.

The location column represents the relative path of the directories from the LINUX_SRC directory. Only seven files among the total of 100 kernel files turned out to have inter code clones. Compared to device drivers, the occurrence rate of inter code clones is one-tenth that of device drivers. Also, the average NBR and RSA values of the seven files were definitely smaller than those of device drivers.

The distribution of the RSA values of the 100 kernel sources is shown in Fig. 2. There is no file whose RSA value is over 0.2. This demonstrates that ordinary kernel sources rarely contain similar codes, in contrast to device drivers.

In this section, we examine inter code clones for ordinary kernel sources. The results show that ordinary kernel sources contain few inter code clones; thus, the similarity analysis among the kernel sources seems to be meaningless. A developer's programming style may affect the existence of inter code clones. That is not to say that the difference between the RSA values of device drivers and those of ordinal kernel sources is only due to the styles of programmers. As Linux is an open source with a long history, most of the codes must be

Table 4. Comparison of gunze.c and mtouch.c sources.

input\touchscreen\gunze.c

```
// … omission

struct gunze {
    struct input_dev *dev;
    struct serio *serio; int idx;
    unsigned char data[GUNZE_MAX_LENGTH];
    char phys[32];
};

static void gunze_disconnect(struct serio *serio)
{
     struct gunze *gunze = serio_get_drvdata(serio);
     input_get_device(gunze->dev);
     input_unregister_device(gunze->dev);
     serio_close(serio);
     serio_set_drvdata(serio, NULL);
     input_put_device(gunze->dev);
     kfree(gunze);
}

static int gunze_connect(struct serio *serio, struct serio_driver
                                *drv)
{
     struct gunze *gunze;
     struct input_dev *input_dev;
     int err;
     gunze = kzalloc(sizeof(struct gunze), GFP_KERNEL);
     input_dev = input_allocate_device();
     if (!gunze || !input_dev) {
               err = -ENOMEM;
       goto fail1;
     }
     gunze->serio = serio;
     gunze->dev = input_dev;
     input_dev->private = gunze;
     input_dev->name = "Gunze AHL-51S TouchScreen";
     input_dev->phys = gunze->phys;
     input_dev->id.bustype = BUS_RS232;
     input_dev->id.vendor = SERIO_GUNZE;
     input_dev->id.product = 0x0051;
     input_dev->id.version = 0x0100;
     input_dev->evbit[0] = BIT(EV_KEY) | BIT(EV_ABS);
     input_dev->keybit[LONG(BTN_TOUCH)] =
                    BIT(BTN_TOUCH);
     input_set_abs_params(input_dev, ABS_X, 24, 1000, 0, 0);
     input_set_abs_params(input_dev, ABS_Y, 24, 1000, 0, 0);

     serio_set_drvdata(serio, gunze);
     err = serio_open(serio, drv); if (err) goto fail2;
     err = input_register_device(gunze->dev);
     if (err)  goto fail3;
     return 0;

     fail3:   serio_close(serio);
     fail2:   serio_set_drvdata(serio, NULL);
     fail1:   input_free_device(input_dev);
              kfree(gunze);
              return err;
}
```

\input\touchscreen\mtouch.c

```
// … omission

struct mtouch {
    struct input_dev *dev;
    struct serio *serio; int idx;
    unsigned char data[MTOUCH_MAX_LENGTH];
    char phys[32];
};

static void mtouch_disconnect(struct serio *serio)
{
     struct mtouch* mtouch = serio_get_drvdata(serio);
     input_get_device(mtouch->dev);
     input_unregister_device(mtouch->dev);
     serio_close(serio);
     serio_set_drvdata(serio, NULL);
     input_put_device(mtouch->dev);
     kfree(mtouch);
}

static int mtouch_connect(struct serio *serio, struct serio_driver *drv)
{
     struct mtouch *mtouch;
     struct input_dev *input_dev;
     int err;
     mtouch = kzalloc(sizeof(struct mtouch), GFP_KERNEL);
     input_dev = input_allocate_device();
     if (!mtouch || !input_dev) {
               err = -ENOMEM;
       goto fail1;
     }
     mtouch->serio = serio;
     mtouch->dev = input_dev;
     input_dev->private = mtouch;
     input_dev->name = "MicroTouch Serial TouchScreen";
     input_dev->phys = mtouch->phys;
     input_dev->id.bustype = BUS_RS232;
     input_dev->id.vendor = SERIO_MICROTOUCH;
     input_dev->id.product = 0;
     input_dev->id.version = 0x0100;
     input_dev->evbit[0] = BIT(EV_KEY) | BIT(EV_ABS);
     input_dev->keybit[LONG(BTN_TOUCH)] = BIT(BTN_TOUCH);
     input_set_abs_params(mtouch->dev, ABS_X,
               MTOUCH_MIN_XC, MTOUCH_MAX_XC, 0, 0);
     input_set_abs_params(mtouch->dev, ABS_Y,
               MTOUCH_MIN_YC, MTOUCH_MAX_YC, 0, 0);

     serio_set_drvdata(serio, mtouch);
     err = serio_open(serio, drv); if (err) goto fail2;
     err = input_register_device(mtouch->dev);
     if (err)  goto fail3;
     return 0;

     fail3:   serio_close(serio);
     fail2:   serio_set_drvdata(serio, NULL);
     fail1:   input_free_device(input_dev);
              kfree(mtouch);
              return err;
}
```

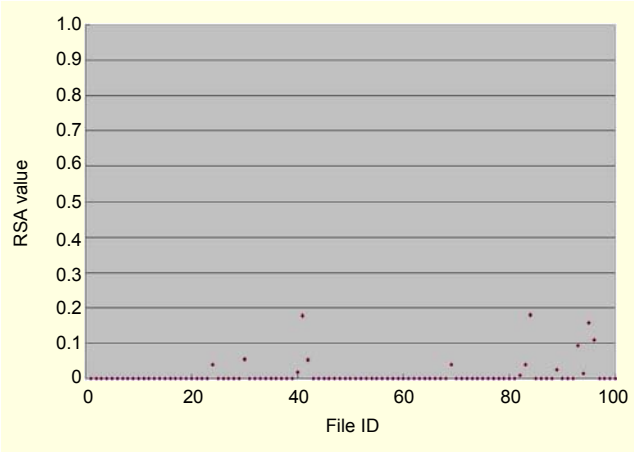

Fig. 2. Distribution of the RSA values of the sources in the LINUX_SRC/kernel directory.

developed as cleverly as possible by experts. Therefore, the occurrence of many inter code clones is an inevitable characteristic of device drivers.

V. Case Study

We conducted a case study to ascertain if our method is useful in the domain analysis of device drivers. This section briefly describes the method and the results of the case study.

1. Method

Writing device drivers requires much time and effort because it requires knowledge of the target device and operating system. Moreover, device drivers of the same domain require the implementation of similar codes, which may be tedious and can lead to errors. To reduce the tedious work, a way to generate such similar codes automatically is needed. Our solution is the use of a template, in which device driver specific information is substituted. To generate a template code, we conducted a similarity study of device drivers of the same domain by analyzing inter code clones.

In this case study, we analyzed inter code clones of the sources of touch screen device drivers in Linux. We chose the touch screen device drivers as a target because their sources show high RSA values and we are familiar with the sources.

Sources of the touch screen devices in Linux are located in the two directories: Linux_SRC/drivers/input/touchscreen and Linux_SRC/drivers/usb/inuput. We searched for inter code clones of these sources by using CCFinder, and then extracted common codes and device driver specific codes from the inter code clones. The template code is constructed from the common codes, and the device driver specific information is substituted.

2. Result and Analysis

Table 5 shows our target files and values obtained with CCFinder. A total of 13 touch screen driver sources are analyzed, and their bus types are diverse (SPI, Platform, RS232, ISA, AC97, and USB). Unlike the NBR and RSA values shown in Tables 1 and 2, which were calculated against whole driver sources, the values shown in Table 5 were calculated against only the 13 touch screen sources. As a result, the NBR and RSA values of Table 5 are relatively small because the number of sources considered was much smaller.

As shown in Table 5, touch screen drivers using the RS232 bus have high RSA values. As shown in Table 3, where the gunze.c and mtouch.c sources are compared, touch screen drivers using the same bus type, RS232, are almost identical without device dependent features, such as name, vendor ID, and product ID.

However, CCFinder reports that inter code clones exist only for touch screen drivers using an RS232. To check whether there is any similarity between touch screen device drivers using different bus types, we compared sources of touch screen device drivers using the USB and RS232 bus types. Table 6 shows part of the usbtouchscreen.c and gunze.c codes. Those codes show substantial differences. In particular, the structures and method names that they use differ. However, although the RSA and NBR values of the usbtouchscreen.c source are zero, there are some inter code clones which were not found by CCFinder. The similar codes are highlighted in Table 6. Evidently, the similarity is mainly from the usage of the input_dev structure. CCFinder

Table 5. Inter code clones among the sources of touch screen device drivers in Linux.

File	Line count	Bus type	NBR	RSA
/touch/ads7846.c	877	SPI	0	0
/touch/corgi_ts.c	371	Platform	0	0
/touch/elo.c	374	RS232	5	0.11
/touch/gunze.c	159	RS232	5	0.56
/touch/h3600_ts_input.c	408	RS232	0	0
/touch/h680_ts_input.c	130	-	0	0
/touch/mk712.c	184	ISA	0	0
/touch/mtouch.c	185	RS232	5	0.61
/touch/penmount.c	150	RS232	5	0.65
/toucn/touchright.c	160	RS232	5	0.71
/touch/touchwin.c	161	RS232	5	0.70
/touch/ucb1400_ts.c	522	AC97	0	0
/usb/input/usbtouchscreen.c	765	USB	0	0
Average	342	-	2.31	0.26

Table 6. Comparison of the gunze.c and the usbtouchscreen.c sources.

\input\touchscreen\gunze.c

```
// … omission

struct gunze {
    struct input_dev *dev;
    struct serio *serio;
    int idx;
    unsigned char data[GUNZE_MAX_LENGTH];
    char phys[32];
};

// … omission



static int gunze_connect(struct serio *serio, struct serio_driver
                                        *drv)
{
    struct gunze *gunze;
    struct input_dev *input_dev;
    int err;

    gunze = kzalloc(sizeof(struct gunze), GFP_KERNEL);
    input_dev = input_allocate_device();
    if (!gunze || !input_dev) {
            err = -ENOMEM;    goto fail1;
    }
    gunze->serio = serio;
    gunze->dev = input_dev;
    input_dev->private = gunze;
    input_dev->name = "Gunze AHL-51S TouchScreen";
    input_dev->phys = gunze->phys;
    input_dev->id.bustype = BUS_RS232;
    input_dev->id.vendor = SERIO_GUNZE;
    input_dev->id.product = 0x0051;
    input_dev->id.version = 0x0100;
    input_dev->evbit[0] = BIT(EV_KEY) | BIT(EV_ABS);
    input_dev->keybit[LONG(BTN_TOUCH)] =
                        BIT(BTN_TOUCH);
    input_set_abs_params(input_dev, ABS_X, 24, 1000, 0, 0);
    input_set_abs_params(input_dev, ABS_Y, 24, 1000, 0, 0);

    // … omission
}
```

\usb\input\usbtouchscreen.c

```
// … omission

struct usbtouch_usb {
    unsigned char *data; dma_addr_t data_dma;
    unsigned char *buffer; int buf_len; struct urb *irq;
    struct usb_device *udev; struct input_dev *input;
    struct usbtouch_device_info *type;
    char name[128]; char phys[64];
    int x, y, int touch, press;
};

// … omission

static int usbtouch_probe(struct usb_interface *intf,
                                const struct usb_device_id *id)
{
    struct usbtouch_usb *usbtouch;
    struct input_dev *input_dev;
    struct usb_host_interface *interface;
    struct usb_endpoint_descriptor *endpoint;
    struct usb_device *udev = interface_to_usbdev(intf);
    struct usbtouch_device_info *type;
    int err = -ENOMEM;

    // … omission (USB related function)

    usbtouch = kzalloc(sizeof(struct usbtouch_usb), GFP_KERNEL);
    input_dev = input_allocate_device();
    if (!usbtouch || !input_dev) goto out_free;

    usbtouch->udev = udev;
    usbtouch->input = input_dev;

    // … omission (USB related function)

    input_dev->name = usbtouch->name;
    input_dev->phys = usbtouch->phys;
    usb_to_input_id(udev, &input_dev->id);
    input_dev->cdev.dev = &intf->dev;
    input_dev->private = usbtouch;
    input_dev->open = usbtouch_open;
    input_dev->close = usbtouch_close;
    input_dev->evbit[0] = BIT(EV_KEY) | BIT(EV_ABS);
    input_dev->keybit[LONG(BTN_TOUCH)] = BIT(BTN_TOUCH);
    input_set_abs_params(input_dev, ABS_X,
                type->min_xc, type->max_xc, 0, 0);
    input_set_abs_params(input_dev, ABS_Y,
                type->min_yc, type->max_yc, 0, 0);

    // … omission (USB related function)
}
```

did not find some inter code clones because it uses a token-to-token matching method to detect code clones. If we use other code clone detection tools that use line-to-line matching, the unfound inter code clones can be detected.

From further investigation of inter code clones of the touch screen device drivers, we conclude that device driver template codes can be generated by the combination of a bus type and a function type. Bus types include RS232, USB, platform bus, and so on. Function types include touch screen, mouse, keypad, keyboard, and so on.

After the bus type and function type are chosen, device dependent information for the types is required. Assume that we are to develop a touch screen device driver that uses an RS232 bus. That is, its bus type is an RS232, and its function type is a touch screen. The device dependent information for the touch screen function type includes name, vendor ID,

product ID, version information, and abs values of the devices.

Based on the above idea, we made a template for touch screen device drivers of the RS232 bus type. With the device dependent values related to the RS232 bus type and the touch screen function type, we can automatically generate about 150 lines of code. When considering touch screen device drivers of the RS232 bus type shown in Table 5, the size of the code generated by our method accounts for more than 50% of the complete sources. It is notable that more than half of the generated code is for the RS232 bus function, and the rest is related to the touch screen function.

VI. Related Works

In the case of operating system code clones, a few studies [14]-[17] have been conducted, mostly targeting the Linux kernel. Godfrey and others [14] conducted a preliminary investigation of cloning among Linux SCSI drivers. They identified clone duplication as the major factor that affects the evolution of the subsystem, and demonstrated that the main source of these clones was in the architecture of the subsystem. Casazza and others [15] used metrics-based clone detection to detect cloned functions within the Linux kernel. They mainly focused on evaluating the extent of cloning in a multi-platform software system supporting driver platforms such as ARM, PowerPC, and MIPS. They concluded that, in general, the addition of similar subsystems is done through code reuse. Antoniol and others [16] conducted a similar study, evaluating the evolution of code cloning in Linux, concluding that the structure of the Linux kernel did not appear to be degrading due to code cloning activities. Li and others [17] analyzed copy paste codes in Linux and FreeBSD and their related bugs.

All of these studies show that there are many code clones in the Linux system, especially for device drivers. The studies also show that the occurrence of code clones in Linux seems to be reasonable as the system evolves or when it supports diverse platform. However, the most important difference between the previous approaches and our approach is that the other approaches analyzed code clones among variations of a specific device driver which occur with different kernel versions. However, our approach analyzes code clones among different device drivers of the same domain under a specific kernel version. Also, although they concluded that the existence of many code clones is a justifiable circumstance, they did not give any suggestion for their possible application.

VII. Conclusion

In this paper, we studied the applicability of code clone detection to analyze the common behavior of device drivers of the same domain. Using CCFinder, a code clone detection tool, inter code clones of device drivers and kernel sources were analyzed. The results demonstrated that many inter code clones exist among device drivers if they are in the same domain. In particular, their sources are strongly similar if they use the same bus type and their inter code clone pairs mainly differ in device dependent information. As inter code clones of device drivers could be reusable components in developing device drivers, the code clone detection method can be helpful for the domain analysis of device drivers.

As a case study, we developed a touch screen device driver template after domain analysis using a code clone detection method. Then, we generated a touch screen device driver from the template code where device dependent information is constituent. The result shows that more than 50% of the codes can be generated for the touch screen device driver using the RS232 bus.

We also demonstrated that the occurrence of many inter code clones is a peculiar feature of device drivers. In the case of kernel sources, few files have inter code clones. However, device drivers show ten times more inter code clones than kernel sources and high RSA values.

The existence of many similar codes among device driver sources of the same domain may be inevitable. However, for this reason, code clone detection can be useful for the domain analysis of device drivers.

The similarity analysis of device drivers can be used in diverse ways. It can be used to design a specification language for them and to generate a template code where device-dependent and system-dependent information can be substituted. It can also be used in the testing of device drivers. Because device drivers of the same domain exhibit common behavior, similar test cases will be repeatedly used for them. Therefore, if we generate common test sets for them in advance, the cost of testing can be reduced by avoiding the generation of the same test sets again and again for each device. Those benefits will increase reusability and reduce redundancy.

References

[1] T. Ball and S.K. Rajamani, "The SLAM Project: Debugging System Software via Static Analysis," *Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, Portland, Oregon, Jan. 16-18, 2002, pp. 1-3.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S.K. Rajamani, and A. Ustuner, "Thorough Static Analysis of Device Drivers," *Proc. of the ACM SIGOPS/EuroSys European Conf. on Computer Systems,* Apr. 18-21, 2006, pp. 73-85.

[3] Y.S. Ma and C. Lim, "Test System for Device Drivers of

Embedded Systems," *Proc. of Int'l Conf. on Advanced Communication Technology*, Feb. 2006.

[4] S.A. Thibault, R. Marlet, and C. Consel, "Domain Specific Languages: From Design to Implementation Application to Video Device Drivers Generation," *IEEE Trans. on Software Engineering*, vol. 25, no. 3, May-June 1999, pp. 363-377.

[5] T. Katayama, K. Saisho, and A. Fukuda, "Prototype of the Device Driver Generation System for UNIX-like Pperating Systems," *Proc. of Int'l Symp. on Principles of Software Evolution*, Nov. 2000, pp. 302-310.

[6] S. Wang and S. Malik, "Synthesizing Operating System Based Device Drivers in Embedded Systems," *Proc. of the 1st IEEE/ACM/IFIP Int'l Conf. on Hardware/Software Codesign and System Synthesis*, Oct. 2003, pp. 37-44.

[7] R. Prieto-Diaz, "Domain Analysis: An Introduction," *ACM SIGSOFT Software Engineering Notes*, vol. 15, no. 2, Apr. 1990, pp. 47-54.

[8] S.C. Chang, A.P.M. Groot, H. Oosting, J.C. van Vliet, and E. Willemsz, "A Reuse Experiment in the Social Security Sector," *Proc. of the 1994 ACM Symp. on Applied Computing*, 1994, pp. 94-98.

[9] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," *Proc. of the Int'l Conf. on Software Maintenance*, Nov. 1998, pp. 368-377.

[10] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE Trans. on Software Engineering*, vol. 28, no. 7, July 2002, pp. 654-670.

[11] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An Empirical Study of Code Clone Genealogies," *Proc. of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering*, Lisbon, Portugal, Sept. 2005, pp. 187-196.

[12] C. Kapser and M.W. Godfrey, "Cloning Considered Harmful," *Proc. of the 13th Working Conf. on Reverse Engineering,* Washington, DC, USA, 2006, pp. 19-28.

[13] B.S. Baker, "A Program for Identifying Duplicated Code," *Proc. of the 24th Symposium on the Interface*, Mar. 1992, pp. 49-57.

[14] M.W. Godfrey, D. Svetinovic, and Q. Tu, "Evolution, Growth, and Cloning in Linux: A Case Study," *In a Presentation at the 2000 CASCON Workshop on 'Detecting Duplicated and Near Duplicated Structures in Large Software Systems: Methods and Applications*, Nov. 16, 2000.

[15] G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. Di Penta, "Identifying Clones in the Linux Kernel," *Proc. of the 1st IEEE Int'l Workshop on Source Code Analysis and Manipulation*, 2001, pp. 90-97.

[16] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta, "Analyzing Cloning Evolution in the Linux Kernel," *Information and Software Technology*, vol. 44, no. 13, Oct. 2002, pp. 755-765.

[17] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Trans. on Software Engineering*, vol. 32, no. 3, Mar. 2006, pp. 176-192.

Yu-Seung Ma received the BS, MS, and PhD degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST), Rep of. Korea, in 1998, 2000, and 2005, respectively. In February 2005, she joined the Embedded Software Development Tool Research Team at the Electronics and Telecommunications Research Institute (ETRI), Rep. of Korea, where she is currently a senior researcher. Her research interests include program testing, mutation testing, and embedded software engineering.

Duk-Kyun Woo received the BS, MS, and PhD degrees in computer science from Hongik University, Rep. of Korea, in 1993, 1995, and 2001, respectively. In January 2001, he joined the Embedded Software Development Tool Research Team at the Electronics and Telecommunications Research Institute (ETRI), Rep. of Korea, where he is currently a team leader and senior researcher. His research interests include compilers, embedded software development tools, and sensor networks.