Visualizing Clone Cohesion and Coupling

Zhen Ming Jiang
University of Waterloo
zmjiang@cs.uwaterloo.ca

Ahmed E. Hassan
University of Victoria
ahmed@ece.uvic.ca

Richard C. Holt
University of Waterloo
holt@plg.uwaterloo.ca

Abstract

Coupling and cohesion between subsystems are commonly studied metrics when analyzing the architecture of software systems. It is usually desirable for subsystems to have high cohesion within the subsystem and to have low coupling to other subsystems. High cohesion implies cohesive concerns and low coupling implies localized changes.

We extend the ideas of coupling and cohesion to code cloning. A code clone is a segment of code that has been created through duplication of another piece of code. Previous research has shown that in some instances code cloning is desirable, whereas in other cases it is not. We believe that it is justifiable to have code cloning within subsystems (high cohesion), whereas it is not justifiable and likely not desirable to have it across subsystems (high coupling). We present an approach, which consists of a framework that generates and filters cloning data and a visualization technique which graphically highlights clone cohesion and coupling between architectural subsystems. Our approach can be used by developers to locate undesirable cloning in their software system. We demonstrate our approach through a case study on the SCSI drivers in the Linux kernel.

1 Introduction

A code clone is a segment of code that has been created through duplication of another piece of code. As much as $10-15\%$ of the source code of large systems is cloned [15].

Cloning is usually considered undesirable for software maintenance. Many approaches advocate removing clones and preventing their introduction by constantly monitoring the source code during its evolution [17]. Code cloning leads to bloated code bases that are difficult to change since modifications (in particular bug fixes) to one piece of code may require propagating the same changes to other clones of the just modified code. Code cloning also increases the cognitive effort needed by developers to understand a large software system, since developers have to examine multiple clone instances in order to tell the difference among them.

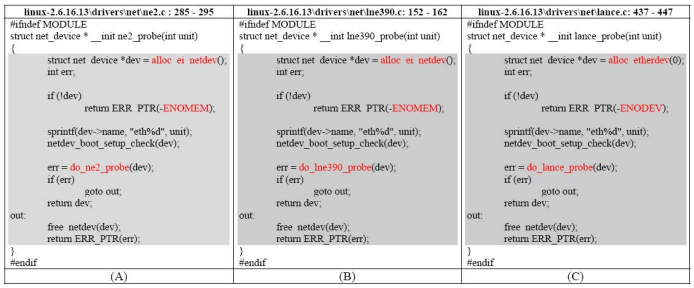
On the other hand, code cloning may be desirable and is sometimes unavoidable. For example, cloning speeds up the development in the short-term. For implementing functions or modules that accomplish similar tasks, cloning serves as a good "mental template" to start with. Moreover copying a critical piece of code and changing it is preferred rather than refactoring to make it more general for reuse. Keeping the critical piece of code unchanged will avoid the introduction of bugs in critical system functionality [10]. In some instances, cloning is actually preferred rather than abstraction for performance considerations [4]. Research has as well shown that cloning cannot be avoided because of the difficulty in abstracting due to programming language limitations [3]. There is also "accidental cloning", which is not caused by direct copy-and-paste but by using the same set of APIs to implement similar protocols [2].

The terms cohesion and coupling are commonly used in studying the design or architecture of a software system. Both terms measure the structure of dependencies within each subsystem and between subsystems (a subsystem contains files or other subsystems). Both terms usually measure static and dynamic code dependencies. Coupling is concerned with dependencies among subsystems; while cohesion refers to the dependencies within the subsystem. It is commonly desirable for a software system to have low coupling and high cohesion. For example, a subsystem with a large number of functions that are dependant on each other (high cohesion) is more desirable over a subsystem where functions depend on functions in other subsystems (high coupling). This intuition forms the basis of many modern software clustering techniques [18].

Highly cohesive subsystems are desirable since they imply that subsystems represent related concerns with a large amount of reuse and similarity between their functionality. Low coupling is also desirable since it implies that they are relatively easy to modify and evolve. Developers changing software systems with low coupling can have their changes focused to a limited number of subsystems instead of needing to propagate their changes to many subsystems.

If we extend the concepts of coupling and cohesion to code cloning, then we believe that it is justifiable to have

XIII ASIA PACIFIC SOFTWARE ENGINEERING CONFERENCE (APSEC'06)
0-7695-2685-3/06 $20.00 © 2006 IEEE



linux-2.6.16.13\drivers\net\ne2.c : 285 - 295

```
#ifndef MODULE
struct net_device * __init ne2_probe(int unit)
{
        struct net_device *dev = alloc_ei_netdev();
        int err;

        if (!dev)
                return ERR_PTR(-ENOMEM);

        sprintf(dev->name, "eth%d", unit);
        netdev_boot_setup_check(dev);

        err = do_ne2_probe(dev);
        if (err)
                goto out;
        return dev;
out:
        free_netdev(dev);
        return ERR_PTR(err);
}
#endif
```

(A)

linux-2.6.16.13\drivers\net\lne390.c: 152 - 162

```
#ifndef MODULE
struct net_device * __init lne390_probe(int unit)
{
        struct net_device *dev = alloc_ei_netdev();
        int err;

        if (!dev)
                return ERR_PTR(-ENOMEM);

        sprintf(dev->name, "eth%d", unit);
        netdev_boot_setup_check(dev);

        err = do_lne390_probe(dev);
        if (err)
                goto out;
        return dev;
out:
        free_netdev(dev);
        return ERR_PTR(err);
}
#endif
```

(B)

linux-2.6.16.13\drivers\net\lance.c: 437 - 447

```
#ifndef MODULE
struct net_device * __init lance_probe(int unit)
{
        struct net_device *dev = alloc_etherdev(0);
        int err;

        if (!dev)
                return ERR_PTR(-ENODEV);

        sprintf(dev->name, "eth%d", unit);
        netdev_boot_setup_check(dev);

        err = do_lance_probe(dev);
        if (err)
                goto out;
        return dev;
out:
        free_netdev(dev);
        return ERR_PTR(err);
}
#endif
```

(C)

Figure 1. Clone example from Linux.

some cloning within a subsystem (clone cohesion); whereas it is not desirable to have cloning across subsystems (clone coupling). Cloning within a subsystem is justifiable since it is likely due to the similarity between functions and files within a subsystem. Large amount of cloning across subsystems is not justifiable since it is expected that subsystems represent different concerns that are not similar and therefore should not share a large amount of code cloning. This intuition is the same as coupling for code dependencies, where it is not desirable to have coupling between different subsystems. In summary, low code coupling and high code cohesion are desirable, and low clone coupling and high clone cohesion are justifiable.

We use the term justifiable for clone coupling and cohesion since as we mentioned earlier that may be good reasons to clone code and there are no definitive research results that rule out the shortcomings or advocate the benefits of clones [16]. Determining whether a clone is desired or not should be done on a project by project basis by system experts. In this paper, we present an approach to assist system experts to study cloning in their software system. The approach presents a visualization that a system expert use to gain an overview of the amount of clone cohesion and coupling in their software system. Using the same visualization, the system expert can investigate specific code clones to determine if they are justifiable or not. If they are not justifiable, then the system expert can schedule their removal as part of future code refactoring activities. The visualization as well permits the system expert to perform "What-if" analysis to determine the impact of removing particular clones and to determine the amount of effort needed to remove clones between subsystems in large software systems.

1.1 Paper Organization

The paper is organized as follows. Section 2 gives an example of code cloning, discusses related work, and highlights limitations of current code cloning visualization work. Section 3 presents our clone extraction framework and discusses the used data schema in our framework. We as well present our visualizations and showcase their main benefits and features. Section 4 demonstrates our visualizations using a case study from the Linux Kernel (in particular its SCSI drivers). Section 5 concludes our paper and outlines future work.

2 Code Cloning

Code clones are identical or near identical fragments of source code. A clone is a segment of code that has been created through duplication of another piece of code. Clones share similar code structures. However, since the size and the degree of similarities among code segments vary, code cloning is a fairly objective concept. It depends on the context whether it is a code clone or not. Figure 1 shows an example of code cloning from the Linux Kernel. The source code is taken from the code responsible for supporting the different network cards in the Linux Kernel version 2.6.16.13. Areas highlighted in grey are clones. Areas highlighted in red are clones variation points.

There are three general techniques to detect clones:

Simple Text Comparison: Simple text comparison techniques try to locate exact matches of code segments. The *Exact Match Clone Detection* algorithm described in [9] is an example of such a technique. The algorithm normalizes the code by removing comments and suppressing white spaces. It then finds all matched lines for each line. The algorithm uses a pattern matching algorithm to generate a list of maximal number of consecutive lines of cloned code for each code segment. Finally, cloning results are generated by filtering out smaller code segments. For the example shown in Figure 1, a simple text comparison technique would not recognize the variation points (in red). Instead of identifying a single large clone code segment, the algorithm would identify several smaller code segments as clones.

Lexical Analysis: Lexical analysis techniques tokenize the code, concatenate tokens into token sequences. Then they create abstract token strings to mark identifers and code constructs. The abstract token strings are used to locate maximal substring matches. An example of a tool that uses such a technique is the CCFinder tool [14]. The example shown in Figure 1 was identified by CCFinder.

AST Analysis: Abstract Syntax Tree (AST) Analysis techniques parse the code and create an abstract syntax tree. The techniques then compare AST subtrees. Clones are detected if two subtrees are identical to each other. An example of such a technique is presented in [4]. The example shown in Figure 1 would be identified by such techinques.

When reporting clones, clone detection tools use two terms: clone pairs and clone classes. A clone pair is a pair of code segments which are identical or similar to each other. A clone pair is both commutative and transitive. Commutative property means that "A clones B" implies "B

clones A". Transitive property shows that if we have clone pairs $< A, B >$ and $< B, C >$, then we have clone pair $< A, C >$. A clone class is the maximum set of code segments in which any two of the code segments forms a clone pair. In this example, $< A, B, C >$ are part of the same clone class.

Referring back to Figure 1, we can see that the probe methods from three different files are almost identical to each other. Right below the source code from left to right, we have marked the source code as A, B, and C, respectively. In this example, a clone detection tool would report 3 clone pairs: $< A, B >$, $< A, C >$ and $< B, C >$, and one clone class, which consists of code segments $< A, B, C >$.

2.1 Related Work on Visualizing Clones

Name	Source Entity	Clone Relation
Scatter Plot [11, 19, 21]	Code Segments	Clone Pair
Metric Graph [21]	Code Segments	Clone Class
File Similarity Graph [21]	File	
Hass Diagram [12]	File	Clone Class
Hyper-linked Web [13]	File	Clone Class
Link Editing [20]	Code Segment	Clone Class
Dependency Graphs [15]	Subsystem	Clone Pair
Duplication Web [19]	File	Clone Pair
Duplication Aggregation Tree Map [19]	Subsystem	Clone Class
System Model View [19]	File, Subsystem	Clone Pair
Clone Class Family Enumeration [19]	File	Clone Class
Our Approach	Subsystem	Super Clone

Table 1. Summary of Clone Visualization

For large software systems, clone detection tools usually report a large number (thousands) of clone pairs and clone classes. In order to help software maintainers in examining the output of clone detection tools, several clone visualization approaches have be proposed in literature. We break down previous research along two dimensions:

1. **Visualized Source Entities**: Are clones shown at the code segment level, lifted to the file level, or lifted to the subsystem level? Higher abstractions (such as subsystems) permit the study of large software systems since they reduce the amount of clutter shown in the generated visualization.
2. **Visualized Clone Relations:** Are clones shown as clone pairs, grouped as clone classes, or grouped as super clones? By grouping clones between common files or subsystems, then practitioners can easily recognize troublesome (large amounts of) cloning between two source entities instead of being overwhelmed by many smaller clone pairs.

Table 1 summarizes current clone visualization research along these two aforementioned dimensions. The table as well compares our presented visualizations to prior work. Our visualizations show clones at the subsystem level, this permits us to scale to handle large software systems. In contrast to previous work, we visualize clones at the clone class or super clone instead of visualizing clones at the clone pair level. We believe that visualizing clones at the clone pair level would increase the complexity of the visualization.

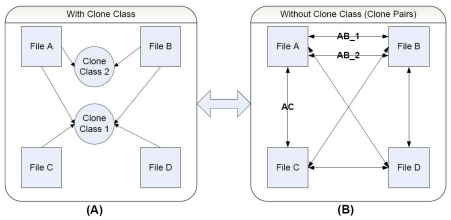


Figure 2. Comparing Both Approaches.

Figure 2 compares two different approaches (visualizing files using clone pairs and clone classes). The square nodes are files, whereas the circle nodes are clone classes Edges indicating cloning relationship. Both views display the same data. The left-hand-side view groups clones into clone classes, whereas the right-hand-side view shows clone pairs. The right-hand-side view contains a large number of cross-cutting edges than the left-hand-side view. These edges make the visualization much harder to view in particular for large software systems. Moreover, the use of clone pairs in the visualization causes the loss of other relevant information. For example, it is not clear that the cloning relationship AB_1 is only between files A and B (clone class 2) or if it is among files A, B, C and D (clone class 1). Both these problems are exaggerated for large software systems.

To further reduce the clutter for large software systems, we propose the creation of super clones which aggregate multiple clone classes between the same source entities into one large super clone. These super clones help highlight and summarize to developers the magnitude of cloning between two source code entities.

3 Our Approach

The main motivation for our visualization is to assist practitioners in coping with the large amount of results displayed by clone detection tools. The main purpose of the generated visualization is to highlight to developers cloning within each subsystem and across subsystems. Developers can then easily investigate whether the cloning is justifiable or not. For example, if a developer examining our visualization notices that there is a large amount of cloning between the memory manager and the network drivers in Linux, then they may be alarmed since there is no clear justifiable reason for such cloning to occur. On the other hand if our visu-

alization highlighted that two similar driver families using the same hardware chipset have a large amount of common code then the developer may consider such cloning justifiable. The developer may later consult other senior developers if such cloning is desirable. In short, our produced visualization simply highlights what we consider to be troublesome clones and we permit developers to study them closely instead of displaying all clones between all code segments in large software systems. We now detail the main two components of our approach: a clone recovery framework and a clone visualization.

3.1 Clone Recovery Framework

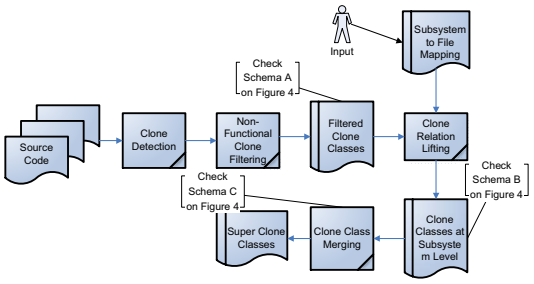


Figure 3. Our Clone Extraction Framework

In order to visualize clones we must first recover them by running a clone detection tool. The results of the clone detection tool is then post-processed in order to remove incorrect cloning relations. We produce our visualizations using the filtered information and domain knowledge (gathered by a system expert or through reading system documentation).

Figure 3 gives an overview of our framework. In order to communicate between the different tools and steps in our framework we used a set of data schemas. Each step in our framework expects data in the appropriate schema. The schemas are shown in Figure 4. The schemas are at varying level of detail: Clone-Class-Code-Segment Level, Clone-Class-Subsystem Level, and Super-Clone-Subsystem Level. The steps in our framework lift the cloning data from **Clone-Class-Code-Segment Level schema**, to **Clone-Class-Subsystem Level schema**; and then merge clone classes to get **Super-Clone-Subsystem Level** cloning relation. We present below the different steps in our framework

3.1.1 Clone Detection

To generate cloning data, we use the CCFinder tool [14] which is reported to have a high recall rate compared to other tools [6].

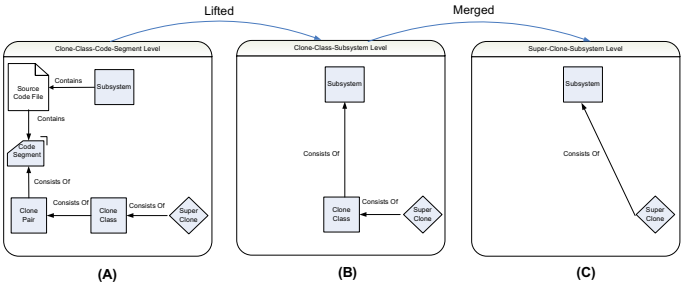


Figure 4. Schemas Used in Our Framework.

In order to reduce the reporting of rather small trivial clones, CCFinder must be configured with a minimum clone size. We chose 30 tokens as the minimum clone size, since previous studies [16, 15] show that the output of CCFinder is of reasonable accuracy at this token level. We also turn off the option to locate clones within the same file, since we are more interested in detecting similarities across source code files and subsystems at the architecture level. Different options can be configured and other clone detection tools by users of our framework if needed. The use of schemas to communicate between the different steps in our framework reduces our dependance on the data format of the different tools, since the output of a tool is converted to our schema and the rest of the steps can easily use the data.

3.1.2 Non-Functional Clone Filtering

Through a manual analysis of the CCFinder output, we noticed that CCFinder occasionally produces incorrect cloning relations. For example, it treats blocks of code that contain variable declarations and function prototypes as clones. For example, the following function prototype declarations taken from $linux - 2.6.16.13/drivers/scsi/aha152x.c$ and $linux - 2.6.16.13/drivers/scsi/esp.c$ are considered as clones by CCFinder:

```
linux - 2.6.16.13/drivers/scsi/aha152x.c:

static void datai_init(struct Scsi_Host *shpnt);
static void datai_run(struct Scsi_Host *shpnt);
static void datai_end(struct Scsi_Host *shpnt);
static void datao_init(struct Scsi_Host *shpnt);
static void datao_run(struct Scsi_Host *shpnt);
```

```
linux - 2.6.16.13/drivers/scsi/esp.c:

static int esp_do_phase_determine(struct esp *esp);
static int esp_do_data_finale(struct esp *esp);
static int esp_select_complete(struct esp *esp);
static int esp_do_status(struct esp *esp);
static int esp_do_msgin(struct esp *esp);
```

We filter out all these non-functional clones in order to reduce the clutter in our visualization and improve the accuracy of our reported results. The filtering is done using a Perl script. The script invokes a source code tagging tool, called ctags [8] then parses the file to determine the beginning and ending line of all defined code entities such

as functions, variables, macros, and prototypes. The script then filters out all identified CCFinder clone pairs which reside outside a function boundary.

The sub-diagram (A) in Figure 4 shows the Clone-Class-Code-Segment Level schema after the clone detection and filtering steps. The data contains four types of entities: source code segments, files, clone pairs and clone classes. A file contains one or more source code segments; each clone pair consists of two code segments; and each clone classes consists of at least one clone pair.

3.1.3 Clone Relation Lifting

At this stage, each clone class contains cloning relations among code segments from different files. We need two lifting operations here. First, we lift information to the file level (i.e. we lift our cloning data from Clone-Class-Code-Segment level to Clone-Class-File level). For example, if clone class A contains lines $110 - 130$ in $file1.c$, lines $210 - 230$ in $file2.c$ and lines $10 - 30$ in $file3.c$, then the lifting will results in clone class A containing 20 lines of cloned code, which reside over files $file1.c$, $file2.c$, and $file3.c$.

Since we plan to visualize relations between subsystems, we need to lift our cloning data to the Super-Clone-Subsystem level. Several clone classes might cross cut the same subsystems or some of them might only contain clones within the same subsystem.

To perform the lifting a file-subsystem mapping is needed. Ideally, such a mapping would be provided by a system expert. However, if we don't have an expert, as suggested in [5], we have to either consult the documentation and group files based on directory structure, naming conventions or manually examine the source code. Continuing the above example, if $file1.c$ is in subsystem $S1$, $file2.c$ in subsystem $S2$, and $file3.c$ in subsystem $S3$; then the lifting result will be clone class A which contains 20 duplicated lines and consists of subsystems $S1$, $S2$ and $S3$.

The sub-diagram (B) in Figure 4 shows the Clone-Class-Subsystem Level schema after the lifting step. The data at this level contains two types of entities: clone classes and subsystems. Each clone classes consists of at least one subsystem.

3.1.4 Clone Classes Merging

In this final step, we aggregate the clone classes which cross cut the same subsystem(s) to simplify the cloning relationship. For example, clone classes 128 and 233 both cross cut subsystem $S1$, $S2$ and $S3$. Therefore, we merge these two clone classes into one super clone node. Super clone nodes are named after the names of the subsystems which they cross-cut. If they cross multiple subsystems, then each subsystem is separated by a "#" sign. In the above example, the super clone is named as $S1\#S2\#S3$ to indicate that they all cross cut subsystems $S1$, $S2$ and $S3$. We found that this naming convention helps easily identify the degree of cloning in a super clone in our visualization instead of having users follow a large number of edges.

The diagram (C) in Figure 4 shows the Super-Clone-Subsystem schema. We have two types of entities here: Super Clone Classes and subsystems: each Super Clone Class consists of one or more subsystems.

3.2 Clone Visualization

The main goal of our visualization is to highlight cloning within each subsystem (cohesion) and across subsystems (coupling). To help direct practitioners to the most troublesome spots, we define "cloning hotspots" that are brightly colored large nodes which grab the attention of the viewer.

A secondary goal of our visualization is to show how different subsystems are interrelated according to cloning. Our cloning visualization shows close together subsystems that have a large amount of common code due to cloning, and shows far apart subsystems that have little cloning between each other.

Node Name	Width	Height
Subsystem (box)	Number of Lines Cloned Within the Subsystems	Number of Clone Classes Involved Within the Subsystems
Super Clone (diamond)	Number of Lines Cloned With Its Associated Subsystems	Number of Clone Classes Involved With Its Associated Subsystems

Table 2. Dimensions of Nodes

Our visualization consists of a graph with nodes and edges. There are two types of nodes in our graphs: rectangle nodes (subsystems) and diamond nodes (super clones). An edge between a rectangle and a diamond represents a cloning relationship.

We now explain the semantics of our visualization by showing how we satisfy our aforementioned goals.

3.3 Goal 1: Cohesion, Coupling and Hot Spots

To highlight cohesion and coupling, we define the dimensions of the nodes representing them according to Table 2. Using these dimensions then large boxes imply that there is a large amount of cloning within a subsystem (high cohesion) and large diamonds indicate that there is a large amount of cloning across subsystems (high coupling).

In addition to varying the sizes of the nodes to highlight the amount of cloning, we vary the color of the nodes. In particular, we vary the color of the diamond nodes since we

0-7695-2685-3/06 $20.00 © 2006 IEEE

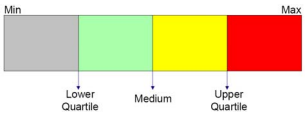


Figure 5. Heat Coloring.

believe that cross subsystem coupling (i.e. high coupling) is troublesome and should be investigated. We consider large diamonds as "cloning hotspots". We "heat color" super clones using a quartile based coloring technique. The color of a diamond is based on the total number of lines cloned across subsystems in that super clone node. We choose the total lines of cloned code rather than the total number of cross family clones since we feel that total lines of cloned code is a better indicator of how much effort will be required to examine and refactor a particular super clone node. The "Heat Coloring" works by calculating the median and the quartiles (the lower quartile is the 25th percentile and the upper quartile is the 75th percentile), the value range of the studied metric is divided into four quarters, which are associated with four different colors respectively. In our case studies, we have chosen red, yellow, light-green, and light-grey as shown in Figure 5.

3.4 Goal 2: Overall System Cloning View

In order to demonstrate the interrelations between different subsystems according to cloning, we apply a force-based graph layout in our visualization to arrange the relative position of subsystems. Weights have been added to the edges to represent the number of cloned lines from the subsystems to the super clone nodes. Subsystems which have more duplications with each other will be placed closer to the super clone classes; conversely, subsystems which have less cross-family cloning will be pushed further away from the super clone nodes. Overall, subsystems which have a large amount of code cloning with other subsystems will be placed closer to these subsystems than other subsystems.

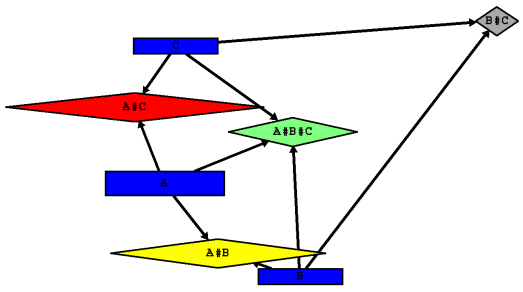


Figure 6. An Example of Our Visualization.

3.5 A Simple Visualization Example

Figure 6 shows an example visualized using our technique. It consists of three subsystems: A, B, and C; and four super clone nodes: $A\#B\#C$, $A\#B$, $B\#C$, and $A\#C$. As indicated by the color: $A\#C$ has the biggest cross-family cloning, subsystems A and C are pulled towards that super clone. The second largest super clone node is $A\#B$, followed by $A\#B\#C$, and finally $B\#C$. Since $B\#C$ is the smallest clones, subsystems B and C are pushed away from that super clone node. Subsystem A is taller and wider, since it contains more internal cloning than other subsystems. All super clone nodes are colored using the heat based coloring techniques.

4 Case Study

To demonstrate our framework, we present a study of cloning in the code responsible for *SCSI* drivers in the Linux kernel. *SCSI* stands for "Small Computer System Interface". We believe studying *SCSI* drivers is a good case study to demonstrate clone cohesion and coupling.

Device drivers are programs for interacting with hardware devices. Studies show that writing device drivers is error-prone and is considered as a major source of errors in operating systems [7]. Around 30% of the source code files in the Linux Kernel are devoted for implementing various device drivers. Due to the similarity between hardware devices in the same family (i.e. from the same vendor or that use the same hardware chipset), developers are more likely to clone code between drivers in the same family in order to speed up development and reduce the likelihood of errors. Therefore, we believe that it is justifiable and probably desirable to have cloning within a driver family. However, it is not justifiable nor desirable to have cloning across different driver families; since such cloning might negatively affect the evolution of Linux. Developers have to be aware of such cross family cloning and may need to propagate changes across driver families. Such change propagation are likely to introduce errors over time as developer forget such unexpected dependencies.

4.1 Results of Our Clone Extraction Framework

All code in the SCSI related directories in Linux consists of $858,727$ tokens, $476,612$ lines, and 381 files. CCFinder (with 30 tokens as the minimum clone size) reported that this code has $54,195$ clone pairs and $2,034$ clone classes. After filtering the non-function clones, we have 305 clone classes left. We have 119 clone classes which cross cut two or more subsystems. We have obtained 33 super clone

0-7695-2685-3/06 $20.00

nodes after the merging process, about 67% of them cross cut two or three subsystems.

4.2 Subsystem Mapping

An important input needed for our framework is the subsystem mapping. Ideally, a system expert would provide such a mapping. Since we did not have access to a system expert, we created a subsystem mapping by reading documentation, analyzing source code and examining files. The subsystem mapping groups files belonging to the same driver family (similar vendor or similar driver chip) in the same subsystem. We explain below how we created our mapping.

There are 425 files that are in the directories that implement the SCSI drivers in the Linux Kernel version 2.6.16.13. Nevertheless, many of these files do not implement drivers instead they are testing or libraries files. For our study we decided to only focus on files that implement specific SCSI drivers. To uncover such files, we started by parsing the Makefiles responsible for building the SCSI drivers in the Linux Kernel. We show below an excerpt of a Makefile for SCSI.

```
obj-$(CONFIG_SCSI_SATA_PROMISE) += libata.o sata_promise.o
obj-$(CONFIG_SCSI_SATA_QSTOR)   += libata.o sata_qstor.o
obj-$(CONFIG_SCSI_SATA_SIL) += libata.o sata_sil.o
obj-$(CONFIG_SCSI_SATA_SIL24)   += libata.o sata_sil24.o
...
obj-$(CONFIG_SCSI_IN2000)   += in2000.o
```

In the above example, $libata.o$ corresponds to a library; and files like $sata\_promise.o$ and $sata\_sil.o$ refer to driver files $sata\_promise.c$ and $sata\_sil.c$, respectively. In addition, $sata\_promise.c$ and $sata\_sil.c$ are part of the same family (i.e subsystem). In order to automate the subsystem mapping process, we followed these steps:

- Object files (e.g. $libata.o$) which appear multiple times are considered as library files and are not considered as driver files. For our analysis all files related to such a library file are considered to be in the same family (i.e. subsystem).
- Object files (e.g $sata\_promise.o$ and $sata\_sil.o$) which appear once in a Makefile are considered as driver files.
- If an object file (e.g $sata\_promise.c$) appears on the same line as a library file (e.g. $libata.o$), then the object file is considered as part of the subsystem called $libata$.
- If an object file (e.g. $in2000.o$) appears alone on line without a library file, then we have to manually examine the file's comment and reading additional system documentation to determine if it is a driver or not and which subsystem it should be mapped to. For example, when we manually examine the $in2000.c$ we find that it is a device driver for the Always IN2000 ISA SCSI card. In grouping such files (ie. files with no libraries), we had two options either to group them according to the vendor or according to their chipset. Each group criteria would result with a different system decomposition, a system expert may decide one criteria over the other. For our study we decided to group such files by chipset in order to be consistent with the grouping created by the Makefile. The grouping obtained from the Makefile, in the previous steps, is based on the chipset.

This process has helped us identify 109 drivers and we created 17 driver subsystems (i.e. families). Our automatic Makefile clustering has helped us cluster 56 drivers. The remaining drivers are clustered manually.

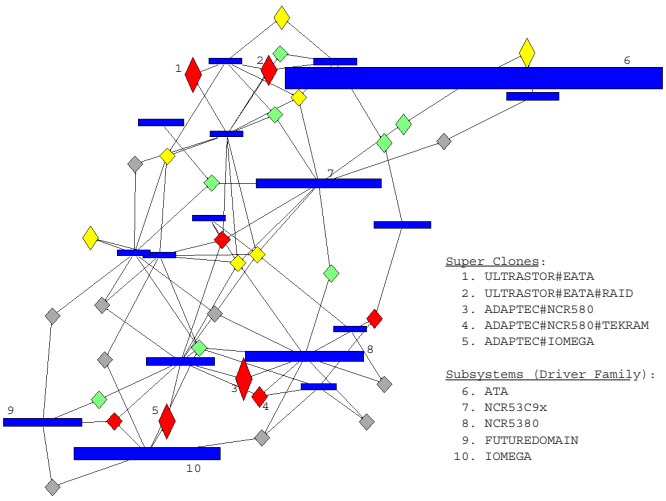


Figure 7. Annotated Screenshot of our Visualization for the SCSI drivers.

4.3 Our Clone Visualization

Figure 7 shows a screenshot of our clone visualization for the SCSI driver subsystems. The figure shows a zoomed out view of the visualization. On the right side of the figure we mark a few noteworthy nodes. The figure is generated using the aiSee tool [1] which permits us to freely navigate through the diagram. The forced-based graph layout permits us to see the degree of clone coupling between subsystems. For example, IOMEGA and FUTUREDOMAIN are more coupled together than ATA and NCR53C9x. In our study we focus on the hot spots (large red diamond nodes and large boxes). Our visualization indicates that there is a large amount of cloning within the ATA and IOMEGA subsystems (ie. both subsystems have high clone cohesion). We investigated as well two of the largest diamond nodes since they indicate high coupling between subsystems).

$ADAPTEC\#IOMEGA$ is one of the biggest super clone nodes (Number 5 in Figure 7). It contains 864 lines of cloned code and has 5 driver files across two subsystems (ADAPTEC and IOEMA). By manually examining

0-7695-2685-3/06 $20.00 © 2006 IEEE

source code, we discover that this super clone node is superfluous. The clones are code segments which contain case switch statements. It is a false clone produced by CCFinder. CCFinder tokenizes the source code and recognizes the similar code structures. However, each case switch statement has the same construct: the case then a following statement.

$ULTRASTOR\#EATA$ is another big super clone node (Number 1 in Figure 7). A closer analysis of this super clone node reveals that it contains 14 clone classes and all the cloning occurs between only two files: $eata.c$ from the EATA family and $u14-34f.c$ from the ULTRASTOR family. They are neither from the same vendor nor do they have a common hardware chipset. $eata.c$ is the Low-level driver for EATA/DMA SCSI host adapters and $u14-34f.c$ is the Low-level driver for UltraStor 14F/34F SCSI host adapters. We decided to explore the reason behind such large degree of coupling between both subsystems (in particular both files). We manually inspected the change logs for both files. The change logs indicate that changes to both files are almost identical and that changes occur almost at the same date throughout the lifetime of both files. Moreover, we discovered that the copyright for both files is attributed to the same person. We suspect that the same developer has cloned one of the files as part of knowledge transfer from one driver to the other. As development progresses, the clones have been maintained synchronously.

The visualization has been able to highlight the most noteworthy clones across subsystems and within subsystems. Using the visualization we are able to quickly locate these noteworthy and investigate them instead of investigating large amount of clone pairs in an ad-hoc manner.

5 Conclusion

A subsystem is a set of modules which are closely related to each other. We believe it is justifiable for cloning to exist within the same subsystem since modules within a subsystem are closely related to each other. Cloning can be considered as a mechanism of reuse. However, cloning across subsystems is usually not desirable nor justifiable. Our framework is used to generate the data need to investigate and manage cloning activities within and across subsystems. We propose a visualization of cloning at the architecture level. We apply our approach to the SCSI drivers in the Linux Kernel. Our visualization directs our focus to the most noteworthy cloning relations in the SCSI drivers.

References

[1] AiSee. http://www.aisee.com/.

[2] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey. Cloning by Accident: An Empirical Study of Source Code Cloning Across Software Systems. In *Proc. of the 2005 Intl. Symposium on Empirical Software Engineering (ISESE-05)*, 2005.

[3] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. Beyond templates: a study of clones in the stl and some general implications. In *In ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 451–459, 2005.

[4] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Tree. In *Proceedings of Sixth Working Conference on Reverse Engineering)*, November 1998.

[5] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, USA, May 1999.

[6] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, pages 36–43, 2002.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 73–88, Banff, Alberta, Canada, Oct. 2001.

[8] Exuberant Ctags. Available online at `http://ctags.sourceforge.net`.

[9] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings ICSM99 (International Conference on Software Maintenance)*, pages 109–118, 1999.

[10] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[11] J. Helfman. Dotplot Patterns: a Literal Look at Pattern Languages. In *TAPOS*, pages 31–41, 1995.

[12] J. H. Johnson. Visualizing textual redundancy in legacy source. In *In Proceedings of CASCON 94*, pages 9–18, 1994.

[13] J. H. Johnson. Navigating the textual redundancy web in legacy source. In *In Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, 1996.

[14] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 28(7):654–670, july 2002.

[15] C. J. Kapser and M. W. Godfrey. Supporting the Analysis of Clones in Software Systems: A Case Study. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2), 2006.

[16] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *ESEC/SIGSOFT FSE 2005*, pages 187–196, 2005.

[17] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Accessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *Proceedings ICSM97 (International Conference on Software Maintenance)*, 1997.

[18] B. Mitchell and S. Mancordis. On the Automatic Modularization of Software Systems Using the Bunch Tool. *IEEE TSE*, 32(3):193–208, 2006.

[19] M. Rieger, S. Ducasse, and M. Lanza. Insights into System-Wide Code Duplication. In *WCRE 2004*, pages 100–109, 2005.

[20] M. Toomim, A. Begel, and S. L. Graham. Managing Duplicated Code with Linked Editing. In *VL/HCC 2004*, pages 173–180, 2004.

[21] Y. Ueda, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Code clone analysis tool. In *Proc. of 2002 International Symposium on Empirical Software Engineering (ISESE2002)*, Nara, Japan, Oct 2002.

0-7695-2685-3/06 $20.00 © 2006 IEEE