

A Framework for Studying Clones In Large Software Systems

Zhen Ming Jiang and Ahmed E. Hassan

University of Victoria
Victoria, BC, Canada
{zmjiang, ahmed}@ece.uvic.ca

Abstract

Clones are code segments that have been created by copying-and-pasting from other code segments. Clones occur often in large software systems. It is reported that 5 to 50% of the source code of a large software system is cloned. A major challenge when studying code cloning in large software systems is handling the large amount of clone candidates produced by leading edge clone detection tools. For example, the CCFinder, clone detection tool, produces over 7 million pairs of clone candidates for the Linux kernel (which consists of over 4 MLOC). Moreover, the output of clone detection tools grows rapidly as a software system evolves. Researchers and developers need tools to help them study the large amount of clone data in order to better understand the clone phenomena in large systems. In this paper, we propose a data mining framework to help researchers cope with the large amount of data produced by clone detection tools. We propose techniques to reduce, abstract and highlight the most interesting data produced by clone detection tools. Our framework also introduces a visualization tool which allows users to query and explore clone data at various abstraction levels. We demonstrate our framework on a case study of the clone phenomena in the Linux kernel.

1. Introduction

A clone is a code segment that has been created through duplication of another piece of code. Clones are quite common in large software systems. It is reported that about 5 to 50% [1, 2, 3] of the source code is cloned. Developers clone code for many reasons. Code cloning may be unavoidable due to language limitations [6] (e.g., for code template [4] or for implementing cross-cutting concerns [7]). Cloning as well helps in minimizing risks [5] by reusing code and designs [4] and experimenting with different designs [8]. However, many researchers believe that cloning is a “bad code smell” since it complicates the maintenance of long lived systems [2, 3, 9, 10, 11, 12]. Unfortunately, there is little empirical study done on the consequence of cloning [13].

One of the major challenges when studying clones in large software systems is handling the large amount of data produced by state of the art clone detection tools. For large software systems such as the Linux kernel (with over 4 MLOC), a clone detection tool (e.g., CCFinder [11]) would produce around 7 million pairs of clone candidates (i.e. potential clones). The volume of clone data reported by CCFinder increases dramatically as the system evolves.

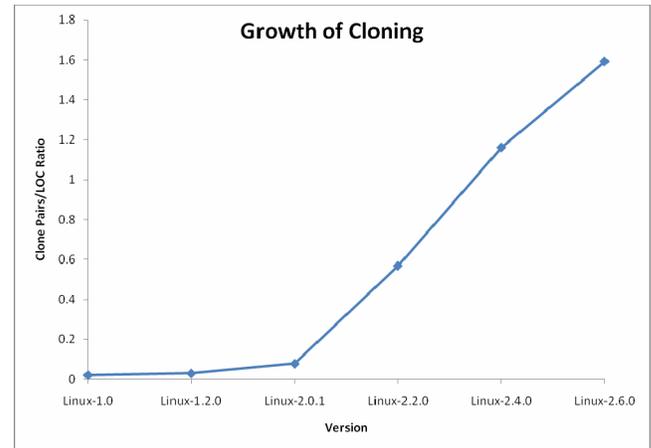


Figure 1. A Plot of the Growth of Clone Pairs As Produced By the CCFinder

Figure 1 plots the number of clone pairs over the lines of code for the Linux kernel over time. The Linux kernel has grown from around 120KLOC (Linux version 1.0) to over 4.0MLOC (Linux version 2.6.0). From the figure, the ratio starts with around 0.02 (Linux 1.0) to 1.6 (Linux 2.6.0). In other words, for Linux 1.0 the amount of clone data exceed by 2% of the size of the code base, whereas the number of clone candidates exceeds by 20% of the size of Linux 2.4.0 and by 40% of the size of Linux 2.6.0. Examining this large number of clone data by hand is not feasible and requires a tremendous amount of effort.

In this paper, we introduce the concept of clone mining. We define **Clone mining** as the process of uncovering interesting clones and clone patterns from the large amount of clone candidates produced by clone detection tools. We define a clone mining framework to support developers and researchers in clone mining activities. Our framework is influenced by traditional data mining frameworks [30, 31]. Whereas traditional data mining frameworks focus on extracting previously unknown or potentially useful information from large data sets, our clone mining framework focuses on uncovering previously unknown or potentially useful clone patterns in large sets of clone data. Our framework uses data reduction and visualization techniques. The framework reduces the clone data by aggregating clone information at various abstraction levels and removing irrelevant data. Our framework also offers a visualization tool which allows users to query and explore clone data at various abstraction levels.

Organization of the Paper

Here is the organization of the paper: In section 2, we give an overview of our clone mining framework. In section 3, we use the Linux Kernel as a case study to apply our framework. In section 4, we present the related work. In section 5, we conclude our paper and propose some future work.

2. An Overview of Our Clone Mining Framework

Figure 2 illustrates our clone mining framework. Our framework consists of four steps: Problem Specification, Data Filtering, Data Reduction and Data Visualization. Here we explain the input to our framework, each step in our framework, and the gained knowledge from using our framework.

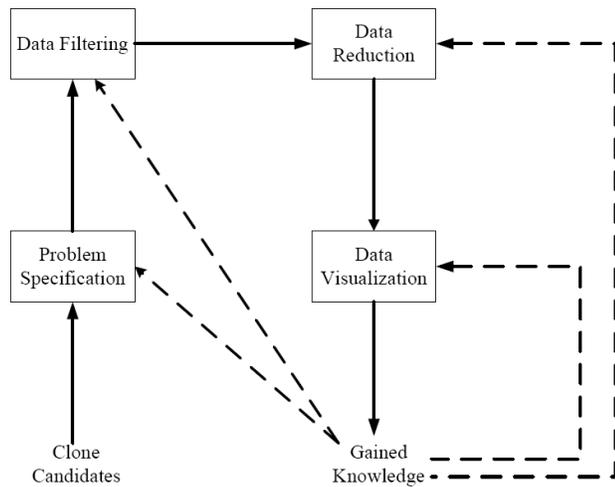


Figure 2. A Framework for Clone Mining

Clone Candidates are segments of code which are reported as clones by clone detection tools. There are several clone detection tools [3, 9, 11, 14]. Each tool uses different heuristics to decide if two code segments are similar (i.e., clone candidates). For example, in metric based clone detection tools [14], code segments which share similar metric value are considered as clones. In abstract syntax tree (AST) based clone detection tools [9], code segments which have the similar AST subtrees are considered as clones. No technique is perfect, as these techniques might miss clones or report false positive clones [15].

The reported clone candidates are represented as either clone pairs or clone classes. A clone pair is a pair of code segments which are identical or similar to each other. A clone class is the maximum set of code segments in which any two of the code segments in the set form a clone pair. For example Figure 3(a) shows 4 clone pairs. The three red code blocks in “File1”, “File2”, and “File3” form three clone pairs. The two blue blocks in “File2” and “File4” form another clone pair. Figure 3(b) represents the same clone information using clone classes instead of clone pairs as shown in Figure 3(a). Figure 3(b) has two

clone classes. A pink diamond indicates a clone class connecting the three red blocks from “File1”, “File2” and “File3”. Another blue diamond represents another clone class connecting the two blue blocks from “File2” and “File4”.

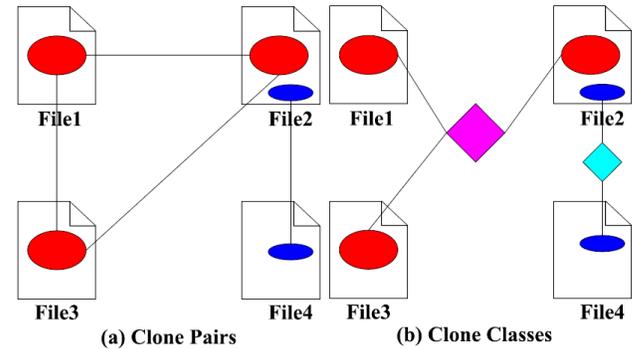


Figure 3. An Example of Clone Pairs and Clone Classes

Problem Specification describes the purpose of our analysis. By defining the purpose of our analysis, we can define *interesting* clones that we should look for (i.e., mine) and not interesting clones which we should prune. The problem specification drives the rest of steps in our framework. For example, we may be interested in studying the spread of the clones. **Spread** refers to the scatter of clones across a software system. Clone pairs or classes may contain files that reside within one subsystem (internal clones) or files which reside in many subsystems (external clones). In particular, we may be interested in only studying widely scattered clones in a code base.

Data Filtering is the operation of removing noise from the data. A fraction of the clone candidates produced by a clone detection tool are false positives. Data filtering is used to remove false positives clones. Filtering techniques depend on the heuristics used by the clone detection tools. For example, the CCFinder tool uses a “Parameterized Token Matching” [11] heuristic to decide whether two code segments are similar. All the reported clones by CCFinder exhibit similar structures. However, the clone candidates may not share similar semantics. “Textual filtering” technique would be suitable for improving the precision of the clone data.

Textual filtering compares pairs of code segments and calculates a textual similarity between them using the diff algorithm. Suppose we have two code segments A and B, then the textual similarity between A and B is defined as $T(A, B) = s/(a+b-s)$, where

a is total number of lines in code segment A;

b is total number of lines in code segment B; and

s is lines of code in which A and B are similar.

If A and B are identical, then $T(A, B)$ is 1. If A and B have nothing in common, then $T(A, B)$ is 0. If the textual similarity is above a threshold, then these two code segments are considered as a clone pair. Otherwise, these two

code segments do not form a clone pair and should be removed from the clone data set. The filtering threshold is determined by trial-and-error in an iterative fashion. In our experiments, we start with a threshold of 0.01 and explore the results of the filtering using a random sampling of 100 clone candidate pairs. Based on the results of the random sampling, the threshold is incremented by 0.01 at each step until the user of the framework is satisfied with the precision of the data.

Data Reduction is concerned with systematically reducing the volume of data. We have used two data reduction approaches: aggregation and pruning. Data aggregation abstracts the clone data to highlight general trends. Data pruning removes the irrelevant clone data to make interesting clone information more visible.

Two data aggregation approaches are introduced here: merging and lifting. We describe both approaches below.

- **Merging:** The merging approach combines clone classes with different line ranges in the same file into a single clone. Also, if two clone classes contain exactly the same set of files or subsystems, then we merge these clone classes into a larger clone class.

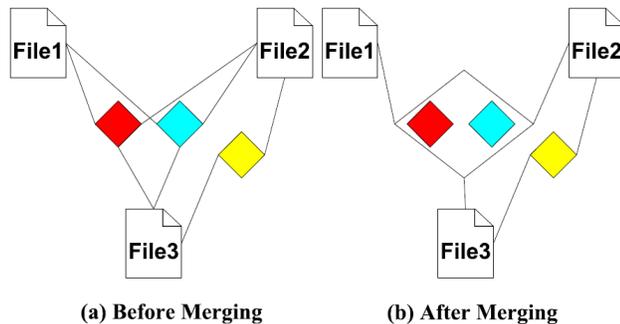


Figure 4. An Illustration of the Merging Approach

We illustrate the merging approach through an example. Figure 4(a) shows three clone classes before merging. The three clone classes are shown in red, blue and yellow. Both the red clone and the blue clone classes contain three files, whereas the yellow clone class contains two files. Figure 4(b) shows the clone classes after merging. We have two clone classes: the big white clone class and the yellow clone class. The white clone class is the result of merging the red clone class and the blue clone class. The yellow clone class remains unchanged since it contains only two files and cannot be merged with the other two clone classes which both contain three files.

- **Lifting:** The lifting approach abstracts clone relations from the code segment level up to the file level or up to the subsystem (i.e., directory) level.

We illustrate the lifting approach through an example. Figure 5(a) shows the clone relations before lifting. We have three directories: *dirA*, *dirB* and *dirC*. Under *dirA*, we have three files: *File1*, *File2* and

File3. We also have *File4* under *dirB* and *File5* under *dirC*. There are two clone classes shown in red and blue respectively. Figure 5(b) shows the result after lifting. Since *File1* and *File2* are both under *dirA*, the red clone class gets lifted to *dirA*. The blue block in *dirA* is from *File3*. The clone relations from *File4* and *File5* are lifted to *dirB* and to *dirC*, respectively.

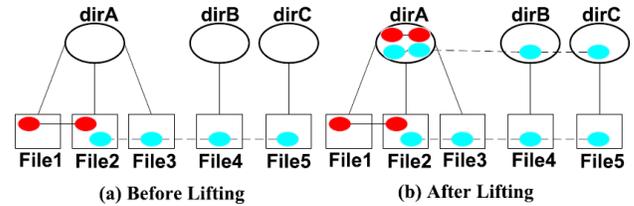


Figure 5. An Illustration of the Lifting Approach

Pruning is the operation of removing uninteresting clones. In this paper, we introduce the notion of interesting clones. Interesting clones are defined in a subjective manner. The definition of interesting clones varies depending on the purpose of a study. For example, if a researcher is interested in examining the clone relations among different filesystems in the Linux kernel (e.g., *ext2*, *nfs*), then a clone in the code implementing the network drivers is not an interesting clone. Or if a researcher is interested in studying clones in a particular subsystem, then all clones not in that subsystem are not interesting clones.

Data Visualization displays the processed data in order to summarize and highlight interesting clones. We have developed various visualization techniques. In [17], we presented the **Clone Cohesion and Coupling (CCC)** graph. The CCC graph highlights the amount of internal clones (clones within subsystems) and external clones (clones across-subsystems) for a single level of system abstraction. In this paper we present another visualization called the **Clone System Hierarchical (CSH)** graph. The CSH graph provides an interactive mechanism for exploring and querying the clone data. The visualization uses a tree layout which mimics the directory structure of the filesystem. We believe that developers are more comfortable working with directories and would be more comfortable exploring clone data using a similar structure. Moreover, the tree layout enables the display of data at different levels of abstractions instead of having to lift the data to a particular abstraction level (as done in the CCC graph). The tree layout shows clones at all levels of abstraction in the same time.

Gained Knowledge refers to interesting clone patterns or clone phenomena uncovered from the clone candidates. Knowledge discovery is an iterative process as shown by the dotted lines in Figure 2. The **Feedback** loop can refine any of our framework steps or specify new problems. In Section 3, we will demonstrate our knowledge discovery framework by studying clones in the Linux kernel.

3. The Linux Kernel - A Case Study

We apply our clone mining framework to study the clone phenomena in the Linux kernel. The Linux kernel is an open source, Unix-like operating system kernel. Linux has been rapidly evolving over the past sixteen years. The total lines of code for Linux has grown at a super-linear rate [20, 21] as the system evolves to add in new features, and to adapt to new hardware platform and devices. The total lines of code started from 176K in version *1.0.0* (March 1994) to 4.6M in version *2.6.16.13* (May 2006). We present our case study following the structure of our framework as shown in Figure 2.

3.1 Clone Candidates

We use the CCFinder tool [11] as our clone detection tool. CCFinder outputs the clone detection results as clone pairs or clone classes. CCFinder is reported to have a high recall and low precision compared to other clone detection tools [15]. We choose 30 tokens as the minimum clone size, since previous studies [26, 27] 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 among subsystems. Different options can be configured and other clone detection tools could be used if needed.

Releases	LOC	Clone Pair	Clone Classes
Linux <i>1.0</i>	118,247	2,486	592
Linux <i>1.2.0</i>	194,794	5,766	1,091
Linux <i>2.0.1</i>	473,190	37,154	3,015
Linux <i>2.2.0</i>	1,114,194	633,522	9,657
Linux <i>2.4.0</i>	2,069,846	2,403,684	19,325
Linux <i>2.6.0</i>	3,626,873	5,773,032	33,000
Linux <i>2.6.16.13</i>	4,601,990	7,369,040	41,064

Table 1. Clones Detected by CCFinder for Linux

Table 1 shows details about the CCFinder output for 7 releases of the Linux kernel. Each row shows the version of the kernel, the total lines of source code (.c and .h files only), the number of clone pairs CCFinder produced and the number of clone classes CCFinder produced. For example, version *1.0* contains 118,247 lines of source code. CCFinder has detected 2,486 clone pairs and 592 clone classes.

3.2 Problem Specification

We are interested in discovering clone patterns in the Linux kernel by studying the spread of clones across the source code for the kernel.

3.3 Data Filtering

We use the textual filtering technique to remove false positive clones produced by the CCFinder tool. After

manually examining the CCFinder output, we find there are mainly two types of false positives.

- One type of false positives is mainly due to similar structure in variable declarations and functional prototypes. Figure 6 shows one example of such functional prototype declarations taken from the *linux-2.6.16.13/drivers/scsi/aha152x.c* and *linux-2.6.16.13/drivers/scsi/esp.c* files. These two code segments are considered as clones by CCFinder.

<i>linux-2.6.16.13/drivers/scsi/aha152x.c:</i>	<i>linux-2.6.16.13/drivers/scsi/esp.c:</i>
static void datai_init(struct Scsi_Host *shpnt);	static int esp_do_phase_determine(struct esp *esp);
static void datai_run(struct Scsi_Host *shpnt);	static int esp_do_data_finale(struct esp *esp);
static void datai_end(struct Scsi_Host *shpnt);	static int esp_select_complete(struct esp *esp);
static void datao_init(struct Scsi_Host *shpnt);	static int esp_do_status(struct esp *esp);
static void datao_run(struct Scsi_Host *shpnt);	static int esp_do_msgin(struct esp *esp);

Figure 6. An Example of a False Positive Clone

- The other type of false positives refers to clones which are similar but are not intentionally cloned by developers. Such clones are called “accidental clones” [29]. Such clones usually occur due to developers having to follow specific protocol or library routines. In the Linux kernel, these types of clones are mainly caused by case-switch statements in the device driver implementations. There are many case-switch statements which follow the format of one case statement followed by one line of method invocation and a break statement. Each case statement normally corresponds to one register, and each register will perform specific actions. The two code segments shown in Figure 7 are similar in structure but have no semantic similarity. The two code segments are taken from Linux *2.6.16.13*. We do not consider these segments as clones since they are not intentionally created by the developers.

<i>linux-2.6.16.13/drivers/scsi/ips.c: 2383-2398</i>	<i>linux-2.6.16.13/drivers/scsi/iscsi_tcp.c: 3466-3477</i>
case IPS_SUBDEVICEID_4M: ha->ad_type = IPS_ADTYPE_SERVERAID4M; break;	case ISCSI_PARAM_IMM_DATA_EN: session->imm_data_en = value; break;
case IPS_SUBDEVICEID_4MX: ha->ad_type = IPS_ADTYPE_SERVERAID4MX; break;	case ISCSI_PARAM_FIRST_BURST: session->first_burst = value; break;
case IPS_SUBDEVICEID_4LX: ha->ad_type = IPS_ADTYPE_SERVERAID4LX; break;	case ISCSI_PARAM_MAX_BURST: session->max_burst = value; break;
case IPS_SUBDEVICEID_5I2: ha->ad_type = IPS_ADTYPE_SERVERAID5I2; break;	case ISCSI_PARAM_PDU_INORDER_EN: session->pdu_inorder_en = value; break;

Figure 7. An Example of An Accidental Clone

We use the textual filtering technique to remove the false positives produced by the CCFinder. Since CCFinder uses a “Parameterized Token Matching” heuristics [11], all the detected clones exhibit similar structures. The false positives and accidental clones are the segments of code which do not have any semantic similarities. We use the amount of common lines between code segments to measure the semantic similarity.

We implemented the textual filtering technique using a Perl script. The scripts reads the clone relations generated from CCFinder and compares the textual differences between the code segments for each clone pair. If the per-

centage of text in common between two code segments falls below a certain threshold, we remove the clone pairs from our analysis. The value of the filtering threshold is determined as follows: we start with the value 0.01. Then we categorize all the clones into different groups with respect to the number of cloned lines (i.e., large, medium, and small clones). We randomly sample a few clone pairs from each of these groups and manually check whether they are false positives. If they are, we set the threshold to be high enough to filter these clones. We notice that our filtering technique also removes the true clones. We repeat this process until we find an optimal value which filters out all the false positive clones in the sample and keeps most of the true clones. For Linux, we use 0.06 as our filtering threshold.

Clearly it is not feasible to verify by hand the accuracy of our filtering technique. We sampled another 100 random clones of the original clone pairs. Such a sample size can be considered enough to ensure a confidence level of 95% and a confidence level of $\pm 7\%$. Our sampling showed that 14 of the clones were false positives not and the filtering has removed 2 of the true clones. So our filtering is reasonably accurate although it is very simple. Table 2 shows the filtering percentage. As we can see our filtering techniques reduces the amount of clone data significantly. Furthermore, as the number of clone pairs gets larger, the textual filtering technique removes more clone data.

Releases	Number of clone pairs		% of filtering
	Before filtering	After filtering	
Linux 1.0	2,486	1,296	47.8
Linux 1.2.0	5,766	1,672	71.0
Linux 2.0.1	37,154	4,583	87.7
Linux 2.2.0	633,522	22,362	96.5
Linux 2.4.0	2,403,684	73,299	96.9
Linux 2.6.0	5,773,032	124,301	97.8
Linux 2.6.16.13	7,369,040	160,707	97.8

Table 2. Results of Our Filtering for Linux

Textual filtering requires many I/O operations since for each clone pair we must process millions of code segments in thousands of files in Linux. Calculations based on our initial implementation for filtering shows that it would takes the implementation more than 2 months to filter the results of CCFinder for Linux 2.6. To address this issue, we re-implemented our filtering in order to minimize the I/O processing. First, we grouped clone pairs by common files so we can compare different code segments from the same pairs of files rather than reading files multiple times. Second we use the Perl's *diff* package rather than the UNIX *diff*. The use of Perl's *diff* enables us to do the textual comparison in-memory rather than writ-

ing the code segments into files and invoking the UNIX *diff* command. These enhancements dramatically improved the performance of our textual filtering implementation. The improved implementation takes 3 hours to perform a full textual filtering of a version in the Linux 2.6 series.

3.4 Data Reduction

Filtering reduces the clone data significantly. However, the clone data set is still large. For example, Linux 2.6.0 contains over 120,000 clone pairs after textual filtering. We perform aggregation and pruning to further scale down the volume of the clone data. We merge clone classes which have the same set of entities into bigger clone classes. We lift the clone classes from the code segment level first to the file level, then to the directory level (from lower level directories to the top level directories). In Linux 2.6.0, the number of relations from the top level subsystems is about 2% of the second level subsystems. The number of clone pairs from the second level subsystems is about 22% of the number of clone pairs from the third level subsystems.

3.5 Data Visualization

Once data is scaled down at various abstraction levels, we visualize the clone data with an emphasis on studying the spread of the clones. **Spread** refers to the scatter of clones across a software system. For a particular clone class how far apart, according to the directory structure, are the cloned files or directories? How many files or directories are in the clone class? Our visualization lays out the clone data in the directory tree structure. It highlights clone relations for individual files and directories by mouse movements.

Knowing the spread of clones at the file level is important because the more spread out a clone is, the more effort is required to modify the code base such as propagating bug fixes selectively to clone instances or to perform re-engineering tasks such as refactoring common code to eliminate clones. For example, *drivers/net/3c501.c* in Linux Kernel version 1.0 has eleven files that have clone relationships. It is relatively harder to maintain than *drivers/FPU-emu/reg_add_sub.c*, which has code duplications with only one file.

Knowing the spread of clones at the subsystem level helps in improving our understanding of the design of the systems. The spread of the clone relations can uncover certain functional relations or clone patterns which are usually not documented. Consider the filesystem subsystem (*fs*) inside the Linux Kernel for example. Subsystems such as *fs/ext2* and *fs/minix* have many external clone relations with each other but have a small number of clones between files inside the subsystems (i.e., internal clones). This is a sign of potential “forking” clone pattern [4]. Forking pattern involves large portions of code duplicates which will evolve independently.

We call our visualization the **Clone System Hierarchical (CSH) Graph**. We first explain the components in our

visualization then we provide an example of using our visualization.

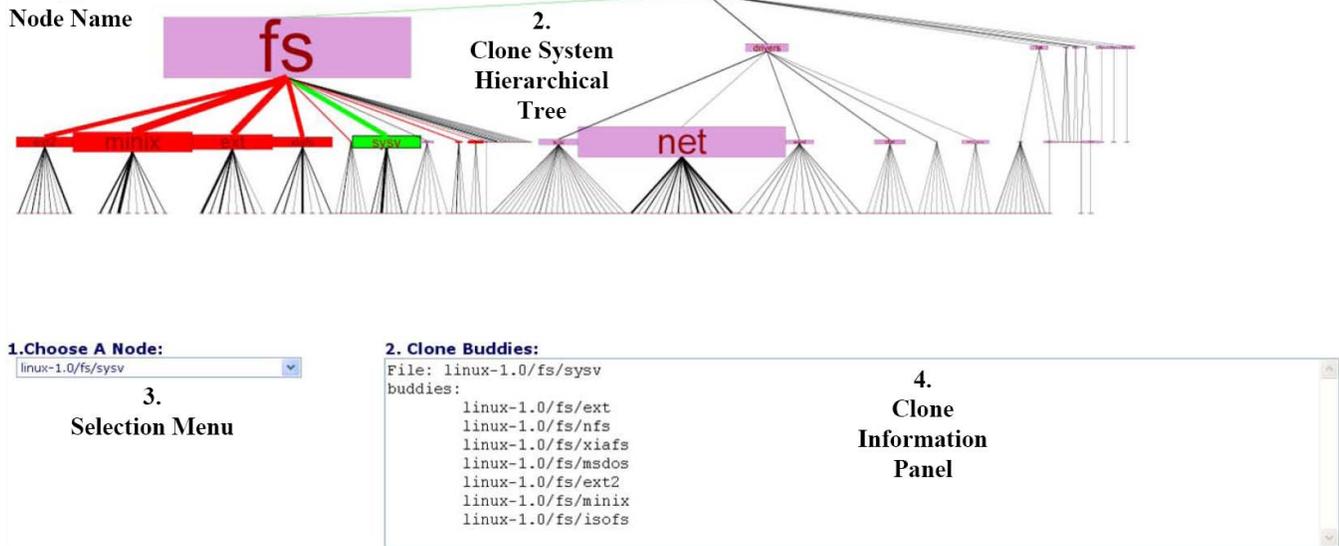


Figure 8. An Annotated Screenshot of the CSH Graph for Linux 1.0

Components of The CSH Graph

The CSH graph consists of 4 components as annotated in Figure 8: **Node Name**, **Clone System Hierarchical Tree**, **Selection Menu**, and **Clone Information Panel**. We explain the details of each component below.

The **Clone System Hierarchical Tree** is an interactive graph which allows a user to select nodes to highlight the spread of clones across the directory structure (i.e., tree) of a software system. Within the tree, we have two types of entities: **nodes** and **edges**. Nodes represent either files (for the lowest level of nodes) or directories (for the rest of the nodes). Edges indicate the containment relationship. For example, there is an edge going from node *fs* to node *minix* indicating that the *minix* is a subdirectory (i.e., subsystem) of *fs*.

Entities	Metric	Description
Node	Width	Number of internally cloned lines within the node (directory or file)
	Height	Number of internal clone classes within the node
Edge	Thickness	Number of cross-subsystem clone classes from this node

Table 2. Descriptions of the Entities in the Clone System Hierarchical Tree

Table 2 describes the metrics embedded in nodes and edges in the CSH graph. The width of the directory nodes shows the number of duplicated lines; whereas the height is proportional to the number of internal clone classes. Flat nodes imply that a subsystem contains very few clone classes but that these clone classes contain a large amount of duplicated code. A thin node indicates that a subsystem contains many small clone code fragments. For file nodes,

the size of the node is constant. We choose to use the same size for the file nodes mainly for scalability concerns: if we embedded the clone information into the dimension of the file nodes it would result in a visualization that is too large to fit in the screen; since there are too many files. The thickness of edges shows the degree of external cloning from that node. The thicker the edge, the larger the amount of external cloning from that node is. Furthermore, sibling directories (directories under the same parent directory) are sorted by the number of their children. The more children that a directory contains, the farther left the directory will be placed. Finally, the graph highlights the clone information for each individual node. Here we define the term: **Clone Buddy**. For example, subsystem A has clone relations with subsystems B, C and D. Then the clone buddies for subsystem A are subsystems B, C and D. In the **Clone System Hierarchical Tree**, when a node is selected, its clone buddies will be highlighted. In addition, we also highlight the edges in order to help researchers trace the selected node and its clone buddies.

Once a node is selected by mouse over or mouse clicking, the **Node Name** component will display the name of the selected node.

The **Selection Menu** allows users to select either a file or a directory to highlight the clone information on the **Clone System Hierarchical Tree**.

The **Clone Information Panel** displays the names of clone buddies for the selected node.

3.6 Gained Knowledge

Interesting clone relations or clone patterns are reported using a combination of our CSH graph and manual examination of the source code. We demonstrate below our

process of extracting some interesting clone patterns from Linux 1.0.

When the CSH graph starts up, it looks similar as Figure 8 except three things: First, the **Node Name** component is blank. Second, all the nodes in **Clone System Hierarchical Tree** remain as pink and edges remain as black. Third, the Clone Information Panel is blank. The initial view gives an overview of the amount of internal and external clones at different directory levels. The visualization indicates the amount of clones by using the size of nodes (for internal clones) and the thickness of the edges (for external clones). Examining Figure 8 we note that the *fs* subsystem has more internal clones than the *drivers* and *net* subsystems given that the *fs* node is larger than the other two nodes. Within the *fs* subsystem, directories like *ext2*, *minix2*, *ext*, *xiafs*, and *sysv* have many external clone relations as indicated by the thickness of the edges. The CSH graph provides an overview of the degree of code cloning within each directory. For example, the *drivers/scsi* directory contains the largest number of files among its sibling directories, thus the *drivers/scsi* (the SCSI device drivers) is placed as the left-most node among all the nodes for the other *drivers* subsystems (all the device drivers). However, the *drivers/scsi* directory contains less clones than *drivers/net* (network device drivers) directory; since the *drivers/scsi* node is small and the thickness of the outgoing edges from *drivers/net* and *drivers/scsi* is about the same.

When a user moves the mouse over a node, the various components in the visualization are updated. First the node's name appears in the **Node Name** component. Also the node's name appears in the **Selection Menu** component. Furthermore, the node, below the mouse, turns green if it has clone relations with other nodes and blue if it does not have clone relations with any other nodes. The clone buddies of the selected node are coloured in red. In addition, the path in the directory tree from the currently selected node up to the root directory will be highlighted in green. Meanwhile, the paths from all node buddies with the selected node up to the root directory are highlighted in red. Finally, the **Clone Information Panel** displays the name of the currently selected node as well as the names of its clone buddies. The information displayed in the panel is helpful in giving the user a textual representation of the results of their query instead of having to follow the different clone edges in the displayed tree.

Figure 8 shows the graph after a user moves the mouse to the node *fs/sysv*: The node name is shown in the upper left corner. The pink node *fs/sysv* turns green. All the clone buddies of the *fs/sysv* node are highlighted in red. The path from the *fs/sysv* node to the root node is coloured green. The paths from the clone buddies of the *fs/sysv* node to the root node are coloured in red. We note that the red path from the *fs* node to the root nodes is coloured in green since the query node and its clone buddies share the same path segment from the *fs* node up to the root node.

The selected node name is displayed in the **Selection Menu**. The **Clone Information Panel** displays the corresponding information: all *fs/sysv* clones are within the *fs* directory. There are no external clones which involve files outside of the *fs* directory. When we move the mouse away from the selected node or click again on the same node; our visualization will undo all the visual changes mentioned above, and revert back to the original state.

In other cases, there may be no clone buddies associated with a node. For example, in Figure 9 we note that there are no external clones for the *drivers/net* subdirectory thus the node is coloured in blue and there is only one path (coloured in green) going from that node up to the root node. Consequently, there are no clone buddies displayed in the **Clone Information Panel**. Many of the nodes under the *drivers* subsystem (e.g., the *drivers/char* node) follow this same pattern: Large number of inner clones and small number of external clones. This pattern is easy to observe in our visualization since the size of the nodes are large (indicating many clones inside them), however the edges coming out from the nodes towards the root node are thin (indication no clone relation with files outside of the subsystem).

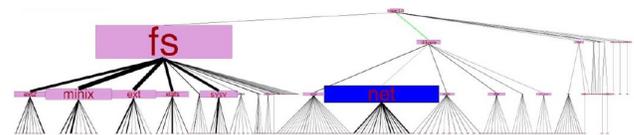


Figure 9. The CSH Graph After Clicking the *net* Subsystem

Pointing and clicking on the graph allows users to navigate and query the clone information interactively. However, certain nodes (like file nodes – the lowest level nodes) may be too small or may be too closely placed to allow the user to pick them correctly. To solve this problem, the drop-down list from the **Selection Menu** can be used to locate nodes. Once we select a node from the drop-down menu, the same visual changes described above will appear as if the node was picked.

The CSH graph enables users to rapidly scroll through the clone data across a file system by using the **Selection Menu**. Once an item from the **Selection Menu** is selected, we can use the up and down arrow keys to move quickly across the various files, this will result in an animation like effect. Since files in the same directory are placed together (items in the selection menu are sorted by file path), then rapidly scrolling through the selection list will help in spotting patterns and anomalies at the directory level. Using the selection menu to rapidly navigate through all the files in the *fs* directory, we notice that all clones occur among files in sibling directories. This is the pattern shown in Figure 10(a). The *fs* directory is the filesystem subsystem in Linux kernel. It contains a number of subsystems which are different types of filesystems like *minix* (*fs/minix*) or *ext2* (*fs/ext2*) or *nfs* (*fs/nfs*). This is an example of the “template” clone pattern [4] where a

subsystem is cloned to serve as a template to create another similar subsystem. The CSH visualization helps in directing our attention to a limited set of files. We then examine these files and note that they all contain a set of structs which have function pointers for each specific operation. For example, the structs in *file.c* are repeated in all types of filesystems which are supported by the Linux kernel (e.g., *vfat*, *nfs*, and *nfs* filesystems). *file.c* contains definitions for structs like *inode_operations* or *file_operations*. *inode_operations* which define interfaces for *inode* related operations. To implement various types of filesystems, developers need to implement functions for reading and writing to a file, creating and removing a file; developer then set the function pointers in the *file.c* file to point to their implemented functions for a particular filesystem. Thus, *file.c* file in one type of filesystem is similar to the *file.c* file in another type of filesystems.

Figure 10(b) shows an anomaly of the above clone pattern. Using the CSH we noticed that the *fs/nfs/mmap.c* file does not have any clone buddies in the *fs* directory, instead the file has clone buddies with the *mm* (memory management) directory. The source code comments of the *fs/nfs/mmap.c* file state that the code is borrowed from *mm/mmap.c* and *mm/memory.c*; which explains the clone relations.

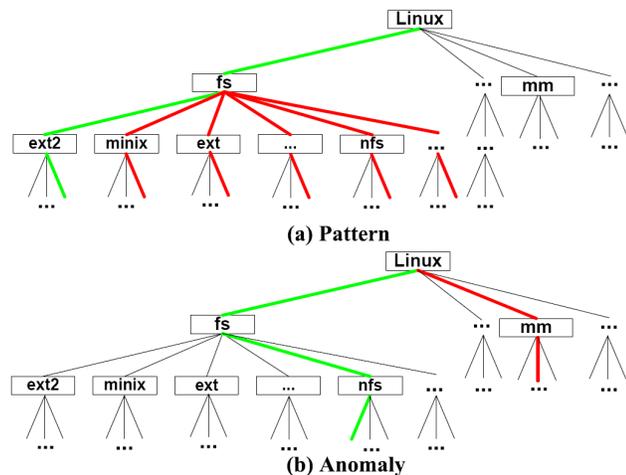


Figure 10. Observed Pattern and Anomaly Using CSH¹

3.7 Summary of the Gained Knowledge

Using our framework to study clones in the Linux kernel, we learned that the *fs* and *drivers* subsystems contain many clone relations. However, these two subsystems exhibit different patterns.

- The *fs* subsystems (e.g., *fs/ext2*) contain more cross-subsystem clones. Each filesystem directory contains a set of files implementing different general functionalities needed for a filesystems (e.g., opening and closing a file). Clones usually happen among files

which implement the same functionality in different types of filesystems.

- There are many clone relations inside *drivers* subsystems (like *drivers/scsi*, *drivers/net*) but few cross-subsystem clone relations. Device drivers which are for similar hardware chipsets tend to have large code segments in common. However, there are few clone relations among drivers of different types.

3.8 Feedback

Having obtained some insight of the clone knowledge inside the Linux kernel, we can either refine the techniques applied in the case study or specify new sub-problems based on the observed clone patterns. Whereas in Section 3.5 we demonstrated the clone patterns observed in Linux 1.0. We tried our visualization on Linux 2.6.0. The amount of clone data is so overwhelming that even the simplest type of clone visualization tool (the scatter-plot from Gemini² [24]) fails to load. As show in Figure 11, the number of nodes is too crowded for humans to extract any interesting patterns.



Figure 11. The CSH Tree for Linux 2.6.0

We apply our framework to prune uninteresting or irrelevant clone information. We use two types of pruning approaches: level pruning and subsystem pruning.

- **Level pruning** removes all the nodes and edges if their abstraction levels are below a threshold value. As we go lower in the directory tree, the number of nodes increases. Figure 12 shows the result of pruning file nodes and the lowest level directories (pruning nodes which are at level 4 or higher). Using the CSH visualization we note that in Linux 2.6.0, the *arch* subsystem has many internal clone relations in comparison to the *fs* subsystem (the left-most directory).

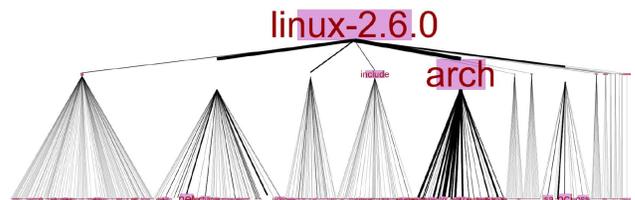


Figure 12. Level-4 Pruning for Linux 2.6.0

- **Subsystem pruning** removes all the clone information which is unrelated to a specific subsystem. If we are interested in clones in the *drivers* subsystem, then all the clone relations not related to *fs* will be removed from the data set.

We can also refine our problem. For example, we can specify a new problem such as studying the number of distinct driver families in the Linux kernel using the clone

¹ Due to space constraints, Figure 10 is redrawn using Microsoft Visio.

² Gemini is the GUI front-end of the CCFinder.

relations. This is an interesting problem to study since drivers are a major source of operating system errors [19] and our case study shows that there are many clone relations among similar drivers but little clone relations among unrelated drivers.

4. Related Work

Basit and Stan apply the Frequent Itemset data mining techniques in order to detect design level clones using the clone information [18]. Cory and Godfrey propose to understand clone in large software systems through categorization [28]. Clones are mapped into eight types of regions (e.g., functions regions and type definitions regions). Several heuristics are used to remove the false positive clones. In contrast, our framework uses a simpler filtering technique. However, Cory and Godfrey's filtering could be adopted into our framework. Kim et al. present an ethnographic study of the developer's copy and paste behavior and propose taxonomy of clone usage patterns [8]. Later, they study the clone evolution for two Java open source projects [16]. In contrast to our automated filtering approach, false positive clones are manually filtered due to the limited size of clones in the ~~Several systems~~ ~~systems~~ have been proposed to visualize clones at the code level, such as the scatter plot [22, 23, 24], the metrics graph [24], file similarity graph [24], Hass diagram [25], Hyperlinked web [26], link editing [27] and exemplar-based visualization [32]. Cory and Godfrey [28] use software architecture like boxes-and-arrows diagram to visualize the clone information. Tairas et al. [33] implement their clone visualization as an Eclipse plugin. It contains both a textual and graphical representation of clone output at the file level. Jiang et al. [17] use force-based graph layout to visualize the amount of clones both within and across subsystems in the Linux Kernel. Rieger et al. [23] propose a number of visualization techniques for qualitatively studying the clone information. Rieger et al.'s System Model View lays out the clone information in a directory structure and embeds clone metrics into the dimensions of nodes and edges. Edges are used to show both containment and clone relationships. In comparison to our visualization, the System Model View shows clone information at single level of the tree instead of showing all the levels of abstractions.

5. Conclusions and Future Work

In this paper, we propose a framework for mining useful clone information from the large data set produced a clone detection tool. Our approach reduces the volume of data by first filtering out false positive clones using a simple lightweight textual filtering technique. The data is scaled down further by aggregating it at various levels of system abstraction. Finally, we use the Clone System Hierarchy visualization to present the data. The visualization is interactive and permits users to explore and query clone data sets using the directory structure of a software system. The directory structure layout since it is familiar

structure for most developers so using the visualization should require a minimal cognitive effort.

In the future, we plan to extend our framework to adopt other data mining techniques such as clustering and association mining. In addition, we want to experiment with different clone detection tools and different filtering techniques. We are as well exploring the use of animation in order to examine the evolution of clones across time. Finally, we would like to perform a user study in order to better understand the strengths and weaknesses of our framework and the CSH graph.

6. ACKNOWLEDGMENTS

The authors thank Dr. Richard C. Holt and Dr. Michael W. Godfrey for their fruitful suggestions on this work.

7. REFERENCES

- [1] Bruno Lague, Daniel Proulx, Jean Mayrand, Ettore Merlo and John P. Hudepohl. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In Proceedings of the International Conference on Software Maintenance, pages 314–321, 1997.
- [2] B.S. Baker. On finding duplication and near duplication in large software system. In Proceedings of Second IEEE Working Conference on Reverse Eng., July 1995.
- [3] S. Ducasse, M. Rieger and S. Demeyer. A language independent approach for detecting duplicated code. In Proceedings of IEEE International Conference on Software Maintenance, August 1999.
- [4] Cory Kapser and Michael W. Godfrey. "Cloning Considered Harmful" Considered Harmful. In Proceedings of the 2006 Working Conference on Reverse Engineering (WCRE-06), Benevento, Italy, October 2006.
- [5] James R. Cordy. Comprehending Reality: Practical Challenges to Software Maintenance Automation. In Proceedings of IEEE 11th International Workshop on Program Comprehension, IWPC 2003 (Keynote paper), May 2003.
- [6] Hamid Abdul Basit, Damith C. Rajapakse and Stan Jarzabek. Beyond templates: a study of clones in the STL and some general implications. In Proceedings of International Conference on Software Engineering, May 2005.
- [7] Magiel Bruntink, Arie van Deursen, Tom Tourwe, Remco van Engelen. An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns. In Proceedings of ICSM 2004, pages 200–209, 2004.
- [8] Miryung Kim, Lawrence D. Bergman, Tessa A. Lau and David Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOP. In International Symposium on Empirical Software Engineering, August 2004.
- [9] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.
- [10] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance*, pages 120–126, 1994.

- [11] Toshihiro Kamiya, Shinji Kusumoto and Katsuro Inoue. CCFinder: A Multi-Linguistic Token-based Code cloning Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.
- [12] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler and M. Bernstein. Pattern matching for clone and concept detection. *Autom. Softw. Eng.*, 3(1/2):77–108, 1996.
- [13] A. Monden, D. Nakae, T. Kamiya and S. Sato and K. Matsumoto. Software quality analysis by code clonings in industrial legacy software. In *Proceedings of IEEE Symposium on Software Metrics 2002*, 2002.
- [14] Kostas Kontogiannis. Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, 1997.
- [15] 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*, pages 36–43, 2002.
- [16] Kim, M., Sazawal, V., Notkin, D., and Murphy, G. An empirical study of code cloning genealogies. *SIGSOFT Softw. Eng. Notes* 30, 5, 187-196. September 2005.
- [17] Zhen Ming Jiang, Ahmed E. Hassan, and Richard C. Holt. Visualizing Clone Cohesion and Coupling. *Proceedings of APSEC 2006: IEEE Asia Pacific Conference on Software Engineering*, Bangalore, India, Dec. 6-8, 2006.
- [18] Basit, H. A. and Jarzabek, S. Detecting higher-level similarity patterns in programs. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT international Symposium on Foundations of Software Engineering*. Lisbon, Portugal, September 05 - 09, 2005.
- [19] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 73–88, Banff, Alberta, Canada, October 2001.
- [20] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the 16th International Conference on Software Maintenance*, pages 131–142, San Jose, California, October 2000.
- [21] Gregorio Robles, Juan Jose Amor, Jess M. Gonzalez Barahona and Israel Herraiz. Evolution and Growth in Large Libre Software Projects. In *Proceedings of the 2005 International Workshop on Software Evolution (IPWSE 2005)*, pages 165 – 174, 2005.
- [22] J. Helfman. Dotplot Patterns: a Literal Look at Pattern Languages. In *TAPOS*, pages 31–41, 1995.
- [23] Matthias Rieger, Stéphane Ducasse, and Michele Lanza. Insights into System-Wide Code Duplication. In *WCRE 2004*.
- [24] Yasushi Ueda, Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Code cloning analysis tool. In *Proc. of 2002 International Symposium on Empirical Software Engineering (ISESE2002)*, Nara, Japan, Oct 2002.
- [25] J. H. Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of CASCON 94*, pages 9–18, 1994.
- [26] J. H. Johnson. Navigating the textual redundancy web in legacy source. In *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, 1996.
- [27] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing Duplicated Code with Linked Editing. In *VL/HCC 2004*, pages 173–180, 2004.
- [28] Cory J. Kapser and Michael 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.
- [29] Raihan Al-Ekram, Cory Kapser, Richard Holt, and Michael Godfrey. “Cloning by Accident: An Empirical Study of Source Code Cloning Across Software Systems”. *Proc. of the 2005 Intl. Symposium on Empirical Software Engineering (ISESE-05)*, Noosa Heads, Australia, 17-18 November 2005.
- [30] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.
- [31] Peggy Wright. 1998. Knowledge discovery in databases: tools and techniques. *Crossroads* 5, 2 (Nov. 1998), 23-26.
- [32] Cordy, J. R., Dean, T. R., and Synytsky, N. 2004. Practical language-independent detection of near-miss clones. In *Proceedings of the 2004 Conference of the Centre For Advanced Studies on Collaborative Research (Markham, Ontario, Canada, October 04 - 07, 2004)*. H. Lutfiyya, J. Singer, and D. A. Stewart, Eds. IBM Centre for Advanced Studies Conference. IBM Press, 1-12.
- [33] Tairas, R., Gray, J., and Baxter, I. 2006. Visualization of clone detection results. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology Exchange (Portland, Oregon, October 22 - 23, 2006)*. eclipse '06. ACM Press, New York, NY, 50-54.