Toward Efficient Aspect Mining for Linux

Danfeng Zhang, Yao Guo, Yue Wang, and Xiangqun Chen
Key laboratory of High Confidence Software Technologies, Ministry of Education
Institute of Software, School of Electronics Engineering and Computer Science, Peking University
Email:{zhdf, wangyue}@os.pku.edu.cn, {yaoguo, cherry}@sei.pku.edu.cn

***Abstract*—Code implementing a crosscutting concern spreads over many parts of the Linux code. Identifying these code automatically can benefit both the maintainability and evolvability of Linux. In this paper, we present a case study on how to identify aspects in the Linux code. First, we analyze four typical crosscutting concerns in Linux and show how to apply existing mining approaches to identify these concerns. We then propose three new mining approaches and compare their performance with the original methods. Experiments show that the proposed mining approaches can find these concerns more efficiently in Linux.**

I. Introduction

Since its introduction in the 1990s, Aspect-Oriented Programming (AOP) [1][11][12] has enhanced the maintenance and evolution of software by separating concerns into modules. In order to successfully apply AOP to existing legacy software, *Aspect Mining* (manually or automatically) is introduced to identify potential aspects from the legacy software. Once aspect candidates are identified, code refactoring mechanisms can be applied to encapsulate these crosscutting concerns into aspects, thus evolving legacy software into aspect-oriented systems.

As a popular open source operating system, Linux has experienced tremendous growth over the last two decades. One important benefit of using Linux is that the developers have total access to the source code: they can modify the source code as they wish (when necessary) to meet specific requirements.

However, to understand the Linux source code is sometimes boring and time-consuming. Furthermore, to modify the source code is typically error-prone, because code that implements one feature/function usually spreads over many parts of the Linux code. These scattered and tangled concerns are called *crosscutting concerns*. One must be very familiar with the source code to be able to modify them.

If we can identify crosscutting concerns in Linux with the help of automated aspect mining tools, it will definitely benefit both the maintainability and evolvability of Linux. Some earlier efforts have tried to apply AOP to Linux [17][18], however, their approaches are mainly focused on how to encapsulate the source code into aspects, i.e. code refactoring, rather than how to find aspects automatically. Furthermore, these approaches tried to find the crosscutting code manually instead of designing tools to identify it automatically.

On the other hand, many automated (semi-automated) mining approaches have been proposed [2]. Most of these work are motivated by Object-Oriented (OO) systems, and few of them target at C language-based systems [9]. As we will show in this paper, these approaches are not efficient enough when applied to Linux. To the best of our knowledge, no research has been reported on automated aspect mining specifically designed for Linux.

In this paper, we first present a case study on how to find aspects in Linux code automatically. We focus on four typical crosscutting concerns in Linux: *parameter check, error handling, synchronization,* and *tracing*. Commonly used automated mining approaches [7][9] are implemented to identify these aspects. However, our experiments show that they are not effective on Linux.

We then present three new approaches to identify the crosscutting concerns in Linux more effectively. The proposed approaches include:

1. A *pattern-based approach* to identify parameter check and error handling concerns.
2. A *Classified fan-in analysis approach* to identify synchronization concern.
3. An *Extended Classified fan-in analysis approach* to identify tracing concern.

We implement the above approaches on top of CDT[*], the C/C++ Development Tools on Eclipse[†]. Evaluations show that the proposed approaches outperform existing approaches considering both coverage and precision.

This paper makes the following main contributions:

- We identify four important crosscutting concerns in Linux, and analyze their properties, especially the symptoms used for mining.
- We present a case study on how current popular mining approaches would fare when applied on Linux. The results demonstrate that current approaches are very inefficient when applied to Linux.
- Three new mining approaches are proposed to improve the aspect mining efficiency on Linux. They have shown great promises toward mining Linux efficiently.

This paper is structured as follows. In the next section, we describe related work. In section III, we identify four important crosscutting concerns in Linux. We discuss how to identify these aspects with existing automated approaches in

This work is supported in part by the National Natural Science Foundation of China under Grant No. 60373001 and the National HighTechnology Research and Development Program("863"Program) of China under Grant No. 2007AA01Z462

[*] www.eclipse.org/cdt/
[†] http://www.eclipse.org/

1530-1362/07 $25.00 © 2007 IEEE
DOI 10.1109/ASPEC.2007.30

section IV. In section V we propose three new approaches to find these concerns more effectively. Then, in section VI, we evaluate these new approaches, comparing them to existing ones. We conclude in section VII.

II. Related Work

Many researchers have adopted AOP to enhance the development and evolution of operating systems. Chen [12] made a survey of research on AOP in operating systems.

Yvonne Coady et al. [13][14][15] conducted a series of research on reverse engineering on Free BSD. The PURE team [16] re-implemented interrupt synchronization in the PURE operating system family with Aspect C++. The Bossa team [18] evolved Linux to support Bossa with AOP. Fiuczynski [17] proposed a tool called c4 to help manipulate patches at the level of their abstract syntax and semantics.

However, all these work on operating systems are focused on how to refactor the systems, rather than how to identify aspect candidates in operating systems. Furthermore, they all identify aspects manually.

On the other hand, many contributions have been made on automated aspect mining for many other applications. Kellens and Mens [2] published a survey of (semi-) automated aspect mining approaches. We classify these approaches by the symptoms of the aspects they considered as below:

A. Identifier Analysis Approaches

Good naming conventions of classes and methods are common in many applications. Some approaches use these conventions to find aspects in code.

Tourwe and Mens [3] proposed a tool called DelfSTof which performs identifier analysis using the Formal Concept Analysis [4] algorithm. Shepherd et al. [5] use Natural Language Processing (NLP) information as an indicator for possible aspect candidates.

B. Fan-in Analysis Approaches

In pre-AOP days, crosscutting concerns were often implemented in an idiomatic way. An example of such an idiom is the implementation of a crosscutting concern by means of a single method in the system, which is called from numerous places in the code.

Gybels and Kellens [6] propose the Unique Methods heuristic that is defined as: "a method without a return value that implements a message implemented by no other method". They detected typical aspects like update notification and memory management in the context of a Smalltalk image. Marin et al. [7] used the fan-in value of a method *m* as the number of distinct method bodies, which can invoke *m*. By identifying high fan-in methods, they found many aspects in a number of open-source Java systems.

C. Clone Detection Analysis Approaches

Another example of an implementation idiom of crosscutting concern in pre-AOP days is "code duplication".

Shepherd et al. [8] use program dependence graphs (PDG) to detect possible aspects. Experiment shows that, this approach can find aspect candidates in reasonable time, and about 90% of the candidates can be refactored. Bruntink et al. [9] made a case study of two other clone detection techniques: token-based, AST-based clone detection in a C-based system ASML. They found that these techniques can find function parameter checking and memory allocation handling aspects efficiently, but it failed in finding error handling and dynamic execution tracing aspects effectively.

III. Crosscutting Concerns in Linux

In this section, we will discuss four typical crosscutting concerns in the Linux code: *parameter check, error handling, synchronization, and tracing*.

A. Properties of Crosscutting Concerns

In the original AOP paper [1], a*spect* is defined as a property that can not be cleanly encapsulated in a generalized procedure, and *crosscutting concern* is defined as a concern that can not be cleanly encapsulated in a generalized procedure.

However, the definition above is not concrete enough to direct aspect mining activities. For example, according to this definition, memory management, interrupt handing, and system calls are all aspects in coarse granularity. In the memory management aspect, page allocation and page swapping are also aspects. And in finer granularity, parameter check and error handling are also aspects. It shows that the granularity of aspect mining is difficult to determine when mining based on this definition.

Marin [10] proposed the concept *crosscutting concern sort* to provide a clearer description of crosscutting concerns. Based on this concept, the crosscutting concerns should have the following properties:

- A general intent,
- An implementation idiom in a non aspect-oriented language, and
- An aspect mechanism to refactor this concern.

These properties are more appropriate to guide aspect mining, because now we understand the purpose (general intent) of the aspect, how to identify it (implementation idiom) and that it is meaningful to identify it (aspect mechanism).

B. Studied Concerns

With the properties above in mind, we choose the following four typical crosscutting concerns in Linux to discuss in detail:

- *Parameter Check Concern:* code to validate a parameter or handle different parameters,
- *Error Handling Concern:* code to check whether a function succeeds, and handle the error accordingly in the case of an error,
- *Synchronization Concern:* code to handle synchronization in Linux, and
- *Tracing Concern:* the trace point in the Linux code implementing the system call "ptrace".

The Linux version we use in our experiments is 2.4.18. Due to the code size limit of CDT, we cannot analyze the

TABLE 1
CODE PERCENTAGES OF FOUR CROSSCUTTING CONCERNS

Aspect	LOC	Fraction
Parameter Check	3943	4.71%
Error Handling	12310	14.69%
Synchronization	1162	1.39%
Tracing	203	0.24%
Total	17618	21.03%

whole Linux system. Instead, we analyzed a subsystem of the Linux system without net, file system, and platform (except i386) related code. The subsystem we analyzed consists of 1064 ".c" files and 83,778 lines of code. We manually marked all places related to these four concerns (during our attempt to analyze the whole Linux code as a special interest group), and calculated the code percentages of each aspect as shown in Table 1.

We choose these four aspects because: (1) parameter check and error handling are very common in Linux, combined making up 19.40% of the code, (2) synchronization plays a very important role in operating systems, and (3) tracing crosscutting concern has a new symptom that no existing mining approach has explored. We will discuss these four concerns in detail next.

1) Parameter Check Concern

Parameter check concern is responsible for two tasks: the validation of parameters, and when a parameter is a flag, this concern is also responsible for handling different flags with dedicated code.

Here is an example for validating a parameter in the file "/linux/kernel/Module.c", where "table" is a parameter of the function "sys_get_kernel_syms":

```
if (table==NULL) {
   unlock_kernel();
   return i;
}
```

The code above checks whether the parameter "table" is NULL, if so, it unlocks the kernel and returns. This code pattern makes up only about 0.5% LOC in Linux.

An example to demonstrate the second task is taken from the file "/linux/kernel/fork.c", where "clone_flags" is a parameter of the function "do_fork":

```
if (!(clone_flags & (CLONE_PARENT|
          CLONE_THREAD))) {
   p->p_opptr = current;
   if ( ! (p->ptrace & PT_PTRACED))
   p->p_pptr = current;
}
if (clone_flags & CLONE_THREAD) {
   p->tgid = current->tgid;
   list_add(&p->thread_group, &current->
          thread_group)
}
```

This code segment checks the parameter "clone_flag", and sets the required field according to the parameter. This kind of parameter check is more common in Linux, making up about 4.2% in Linux code.

Although we have shown only two simple examples to demonstrate this concern, this crosscutting concern is very important to Linux because it makes up almost 5% of Linux source code as shown in Table 1.

2) Error Handling Concern

Error handling crosscutting concern is the code used to check whether a function succeeds, and handle the error in case of an error.

Here is an example, which is taken from the file "/linux/kernel/fork.c", in function "do_fork":

```
p = alloc_task_struct();
if (!p)
   return p;
```

This code segment calls the function "alloc_task_struct", and checks its return value *p*. If it is NULL, it indicates that some error has occurred. This code pattern shows a simple example of the error handling concern. The pattern and the called function may vary for different occurrences of this concern.

As shown in Table 1, this crosscutting concern makes up almost 15% of Linux source code. It suggests that, if we can refactor the kernel by modeling this concern separately into an aspect, we can reduce up to 15% of the main routine source code. Separating this crosscutting concern can greatly improve the readability and maintainability of the Linux code.

3) Synchronization Concern

Synchronization crosscutting concerns are responsible for handling synchronization in Linux.

Many synchronization mechanisms are used in Linux 2.4.18, such as atomic operation, mutex, read/write semaphore, spin lock, read/write lock, and big kernel lock. Functions related to these mechanisms are summarized in Table 2, where the bolded functions are function-like macros, and others are inlined functions. As defined by GNU[‡], a function-like macro is a macro "whose use looks like a function call", and an object-like macro is "a simple identifier which will be replaced by a code fragment". The calls of these certain function-like macros or inlined functions are the symptoms of synchronization aspect.

Synchronization crosscutting concern in PURE has been discussed in [16]. In Linux, identifying this crosscutting concern is important because:

1. Many main routine code in Linux depends on the implementation of the synchronization scheme. It is error-prone to take a new synchronization scheme in Linux code. As pointed out in [16], encapsulating these code can greatly improve the architectural flexibility for an OS.

2. The synchronization code itself depends on the synchronization primitives. It is hard to understand and modify these code without modeling the usage and implementation of these synchronization primitives together.

As shown in Table 1, this crosscutting concern makes up

[‡] http://gcc.gnu.org/onlinedocs/gcc-4.1.0/cpp/

TABLE 2
SYNCHRONIZATION MECHANISMS IN LINUX 2.4.18

Mechanisms	Related functions
Atomic operation	**ATOMIC_INIT**, **atomic_read**, **atomic_set**, *atomic_add, atomic_sub, atomic_dec, atomic_add_negative, atomic_sub_and_test, atomic_inc, atomic_dec_and_test, atomic_inc_and_test*
mutex	**DECLARE_MUTEX**, **DECLARE_MUTEX_LOCKED**, *down_interruptible, init_MUTEX, init_MUTEX_LOCKED down_trylock,up,down,*
read/write semaphore	**DECLARE_RWSEM**, *init_rwsem, down_write, up_read, up_write down_read, rwsem_atomic_update, rwsem_atomic_add,*
spin lock	*spin_lock, spin_trylock, spin_unlock*, **spin_lock_init, spin_is_locked**, **spin_unlock_wait**, **spin_lock_bh, spin_lock_string,spin_unlock_string, spin_lock_irqsave**, **spin_lock_irq**, **spin_unlock_irqrestore**, **spin_unlock_irq**, **spin_is_locked spin_unlock_bh**, **spin_trylock_bh,**
read/write lock	**read_lock, write_lock, read_unlock, write_unlock, rwlock_init**
big kernel lock	*lock_kernel, unlock_kernel,* **kernel_locked, release_kernel_lock, reacquire_kernel_lock**

1.39% of Linux source code.

4) Tracing Concern

Tracing crosscutting concern is an example to show how AOP can enhance the development of software. In Linux, tracing crosscutting concern is the tracing point implementing the system call: "ptrace".

Here is an example taken from the function "do_fork" in the file "/linux/kernel/fork.c":

```
if (p->ptrace & PT_PTRACED)
    send_sig(SIGSTOP, p, 1);
```

Such tracing stubs are used to implement the system call "ptrace", which is used to trace the system at runtime. Providing these tracing points can offer a powerful debugging and profiling tool to both OS developers and users. For example, DTrace in Solaris [20] is a more powerful implementation of tracing. In DTrace, even the smallest system allows as many as 30,000 instrumentation points to be traced.

As shown in Table 1, there are only 203 tracing points in the subsystem we analyzed. Although the number is significantly lower compared to DTrace, identifying this crosscutting concern in Linux is still beneficial because:

1. It will be clear where these tracing points are and what "ptrace" can do if we encapsulate this concern clearly.

2. By separating this crosscutting concern, it will be much easier and clearer to add new tracing points in Linux. This will definitely enhance the evolvability of Linux. With the help of AOP, we could develop a more powerful tracing tool on Linux.

IV. CURRENT ASPECT MINING APPROACHES

As mentioned in section II, many aspect mining approaches have been proposed. We will implement some of the current mining methods for each of the four crosscutting concerns, and evaluate their mining efficiency in this section.

A. Evaluation Metrics

We first introduce two metrics to evaluate each aspect mining approach:

1. *Coverage*: the percentage of places can be identified which is related to a certain crosscutting concern. It is a metric specific to a certain concern.
2. *Precision*: the percentage of identified aspect candidates which are "true" aspects. If an approach can find more than one type of concerns, all related concerns identified must be considered to calculate its precision.

We can see that a low coverage requires a lot of work to find other occurrences of the concern in the code, while a low precision requires extra work to filter out "false" aspects candidates. Our ultimate goal is to develop a good aspect mining approach that will achieve both high coverage and high precision.

B. Parameter Check and Error Handling Mining

Bruntink *et al.* [9] have tried to find three similar crosscutting concerns in a C language system called ASML with clone detection. These three concerns are called: "*General error handling and administration", "Function pre and post condition checking" and "Dedicated handling of errors originating from C memory management*". They found that clone detection can efficiently identify concerns such as parameter checking and memory error handling.

We used CCFinder [21] (version 10.1.12.4), the same clone detection tool used by Bruntink *et al.* [9], to evaluate its performance on the Linux code. We set all the configurable setting as used in [9].

CCFinder found a total of 4303 pairs of clone code; we randomly analyzed 200 of these pairs carefully, and found that among these 200 pairs, 117 pairs lead to a crosscutting concern. Based on this, we estimate that the precision of this approach in Linux is roughly 58%.

Among 734 places that are related to error handing aspect, CCFinder can find only 324 of them, so the coverage for error handling aspect is 44.14%. Similarly, we find the coverage for parameter check aspect is 46.89%.

Neither the precision, nor the coverage seems encouraging. Although this semi-automatic aspect mining approach can find about 45% of these two crosscutting concerns, we have to find the others manually, and filter out about 40% "false" aspect candidates as well. Considering the importance of these two crosscutting concerns in Linux as discussed in section III, we believe a better approach should be taken for these two crosscutting concerns.

C. Synchronization Mining

Fan-in analysis [7] is motivated by the symptom of a lot of function calls in the source code. It is exactly the symptom of synchronization crosscutting concern as we discussed in section III. Thus it might be a good idea to find this aspect using the fan-in analysis approach. As proposed in [7], the *fan-in* of a method *m* is the number of distinct method bodies that can invoke *m*. This approach follows three steps:

> *Step 1.* Automatically compute the fan-in metric for all the methods in the targeted source code.
>
> *Step 2.* Filter the result of the first step:
>
> - Restrict the set of methods to those having a fan-in above a certain threshold.
> - Filter getters and setters from this restricted set.
> - Filter utility methods such as *toString()*.
>
> Step 3. (Mainly manual) Analysis of the remaining set of methods.

To adapt this approach in Linux, we notice that:

1. Because there is no polymorphism in Linux, the calculation of *fan-in* will be much simpler than OO systems. The fan-in is just the number of times this method was called throughout the source code.

2. Besides functions, there are also function-like macros acting like functions in C. To solve this problem, we extend the definition of fan-in to include the fan-in of function-like macros.

3. Getters and setters are a little different in Linux: they have prefixes of *set_, get_,* and we can use this clue to filter them. Although there are no utility methods such as *toString()* in Linux, there are some meaningless functions, such as functions with a prefix of *_do,* which needs to be filtered.

According to the above discussions, we implemented a tool as a plug-in on Eclipse to evaluate the fan-in analysis performance in Linux. We used the indexing interface provided by CDT to parse the code, and get all the functions and function-like macros. Using the search interface provided by CDT, we can get the fan-in metric of all the functions.

To figure out which threshold is better for Linux, we evaluated all the results with a threshold ranging from 20 to 140, with an interval of 20. Manually, we find 1162 places of synchronization concern in the Linux code examined, while this approach can find only 298 occurrences in the code at a threshold of 60. A detailed result can be found in Figure 2.

The figure shows that a threshold of 60 is the best when considering precision. But at this threshold, the coverage is only 26.65%. With a lower threshold, we may get a higher coverage, but with a lower precision. We will propose a more specific approach in section V.

D. Tracing Mining

We found no previous work on mining tracing concern in Linux. Bruntink *et al.* [9] used clone detection to find dynamic execution tracing, but this concern is not like that in Linux, and their evaluation showed that clone detection is not suitable for this concern.

We notice that fan-in analysis is not suitable for this concern, because there are no high fan-in function calls.

We also tried to use clone detection on it. We can only find 5 places of tracing crosscutting concerns out of the total of 41 concerns in the source code with CCFinder. The coverage is only 12.19%.

To find this concern, we will propose a new aspect mining approach using macros in C language.

V. New Mining Approaches

As shown by the above experimental results, although some proposed approaches can be used to find the four different crosscutting concerns, the performance is very poor. In this section, we introduce three new approaches to find these four aspects more efficiently.

A. Parameter Check and Error Handling Mining

As discussed in the previous section, clone detection works poorly for these two concerns. Through examining the source code, we found each concern follows a certain code pattern.

Parameter check follows a specific code pattern that can be summarized as the following production rules, where terms ***if, lpar, rpar, else, switch, lbpar, rbpar, null*** in bold stands for terminals *if, (,), else, switch, {, },* and *null*, and non-terminator *exp_of_para* stands for an expression of a parameter or a field of a parameter, *code_segment* stands for a segment of code, and so on.

> parament_check → **if lpar** exp_of_para **rpar**
> code_segment else_segment |
> **switch lpar** exp_of_para **rpar lbpar**
> case_statements default_statement **rbpar**
> else_segment → **else** code_segment | **null**

Similar to parameter check aspects, error handling also follows the production rules below. Where non-terminator *exp_of_funcall* stands for an expression of a function call, and *branch_statement* stands for if-else statement or switch statement.

> error_handing → **if lpar** exp_of_funcall **rpar**
> code_segment else_segment |
> **switch lpar** exp_of_funcall **rpar lbpar**
> case_statements default_statement **rbpar** |
> assign_statement statement branch_statement
> assign_statement → id **EQ** function_call **semicolon**
> else_segment → **else** code_segment | **null**

Because each of these two concerns follows a certain code pattern, i.e., a symptom, we can use a *pattern-based approach* to find such aspects. This approach matches the exact code patterns to identify aspect candidates, and it is suitable for aspects with a specific pattern, such as the parameter check and error handling crosscutting concerns in Linux.

We use DOM (Document Object Model) generated by CDT to match the pattern defined by the user. By walking through the DOM tree, we match the two patterns above.

This approach can also be used to find other crosscutting concerns with a proper code pattern. An expert who is

familiar with the source code will be needed to define the code pattern. We will compare the performance of this approach with the clone detection method in section VI.

B. Synchronization Mining

To understand why fan-in analysis works poorly for synchronization concern in Linux, we analyzed the result of fan-in analysis approach, and found out that only a few of these functions (or function-like macros) related to synchronization have a high fan-in in the code (such as *spin_lock()*). In order to find more synchronization functions, i.e. to improve its coverage, the threshold should be smaller. However, a smaller threshold might bring more "false" aspect candidates, resulting even worse precision.

We notice that Linux has a well-followed naming convention: functions related to the same synchronization mechanism that should be encapsulated in a same aspect usually have the same prefix, such as *automic_, spin_*. With this clue, we could classify these functions (or function-like macros) according to their prefixes, and calculate the fan-in of a whole cluster of functions. Functions in the same cluster usually belong to a same concern that should be encapsulated in an aspect. It is more meaningful to use the fan-in of a cluster, rather than the fan-in of a single function.

As discussed above, a *cluster* of methods is defined as a collection of methods with the same prefix; *Fan-in* of a *cluster c* is defined as the sum of *fan-in* of each method *m* that belongs to *c*.

We propose a *classified fan-in analysis* approach which takes the following four steps:

Step 1. Classify all the functions and function-like macros into classes by the prefix of their signature.

Step 2. Automatically compute the fan-in metric for all the classes generated in step 1.

Step 3. Filter the results of step 2:

- Restrict the set of classed to those having a fan-in above a certain threshold.
- Filter getters and setters from this restricted set by filtering out classes with a prefix *get_*, *set_*.
- Filter meaningless classes, such as classes with a prefix *_do*.

Step 4. (Mainly manual) Analyze methods in the remaining set of classes.

By using the search interface provided by CDT, we can get the fan-in of the certain prefix, i.e., a cluster of function or function-like macros.

Although this approach is motivated by finding synchronization concern in Linux like other fan-in analysis approaches, it could find all the aspects with this same symptom. We will show what typical concerns besides synchronization concern it can find in section VI. We will also evaluate the performance of this approach in comparison with the original fan-in method in section VI.

C. Tracing Mining

As discussed in section IV, no existing approaches were designed to find the symptom of this concern in Linux.

After analyzing the code carefully, we find out that the tracing aspect always contains object-like macros that start with "*PT_*". All these macros are defined in the file "\linux\include\linux\Sched.h":

```
#define PT_PTRACED          0x00000001
#define PT_TRACESYS         0x00000002
#define PT_DTRACE           0x00000004
#define PT_TRACESYSGOOD 0x00000008
#define PT_PTRACE_CAP   0x00000010
```

The appearance of these object-like macros is the unique symptom of tracing aspect. We propose an *extended classified fan-in analysis* for this concern. Similar to the discussion above, to find a cluster of object-like macros is more reasonable than to find a single macro.

Here, we define a *cluster* of object-like macros as a collection of object-like macro with same prefix, and the *fan-in* of a *cluster c* as the sum of *fan-in* of each object-like macro *o* that belongs to *c*.

This approach can be described as four steps:

Step 1. Classify all the object-like macros into classes by the prefix of their signature.

Step 2. Automatically compute the fan-in metric for all the classes generated in step 1.

Step 3. Filtering of the result of step 2:

- Restrict the set of classed to those having a fan-in above a certain threshold.
- Filter meaningless classes, like class with a prefix *MAX_, MIN_*.

Step 4. (Mainly manual) Analyze object-like macros in the remaining set of classes.

The differences of this approach compared to the previous *classified fan-in analysis* include:

1. What we analyze here are object-like macros, not function-like macros or functions in *classified fan-in analysis.*

2. These two approaches are motivated by different aspect symptoms. *Classified fan-in analysis* is based on the implementation idiom of a single method in the system which is called from numerous places in the code. Its motivation is the same as fan-in analysis approach proposed in [7]. While this approach is based on the phenomenon that many aspects in Linux can be found by certain object-like macros, like *PT_* prefix for trace aspect. It can find many crosscutting concerns that can not be found by *classified fan-in analysis*.

Like *classified fan-in analysis* approach, this approach can also find all the aspects with the same symptom. We evaluate the performance of this approach in the next section.

While implementing this approach, there is a small difference from the previous classified fan-in analysis. Because CDT has replaced all object-like macros in the preprocessing pass before indexing, we use a text-based search for object-like macros instead.

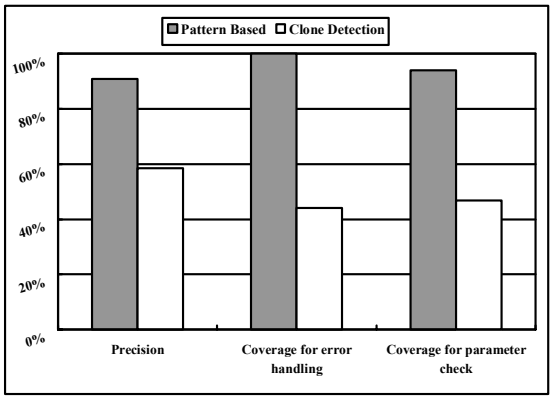


Figure 1. Performance Comparison Between Pattern-Based Approach and Clone Detection.

VI. EVALUATION

We implement our new approaches as a plug-in on Eclipse. We evaluate the performance for each concern in this section.

A. Parameter Check and Error Handling Mining

Among the aspect candidates found by the pattern-based approach, 691 places are related to the error handing aspect. In contrast, we manually find 734 places related to error handing. So the coverage of the pattern-based approach for error handling is 94.14%.

Because parameter check crosscutting concern follows the pattern we described strictly, it is not surprising that the pattern-based approach can find all the 1954 places we manually marked. Thus the coverage of pattern-based approach for parameter check is 100%.

Among the 2347 results (some of them are both parameter check and error handling), 2134 of them are real concerns. So the precision of this approach in Linux is 90.92%.

We show the comparison of pattern-based approach and clone detection in Figure 1. Compared to clone detection, this new approach can achieve both higher coverage and higher precision.

B. Synchronization Mining

Manually, we find 1162 occurrences of synchronization crosscutting concerns in the Linux code studied, while classified fan-in analysis can find 1121 occurrences at a threshold of 20, and 827 occurrences at a threshold of 60. On the other hand, with the original fan-in analysis approach, it could only find 620 occurrences at the threshold of 20, and 298 occurrences at the threshold of 60.

We show a comparison of the coverage of synchronization crosscutting concern between these two approaches at different threshold in Figure 2(A). It is obvious that the classified fan-in analysis approach improves the coverage of synchronization significantly.

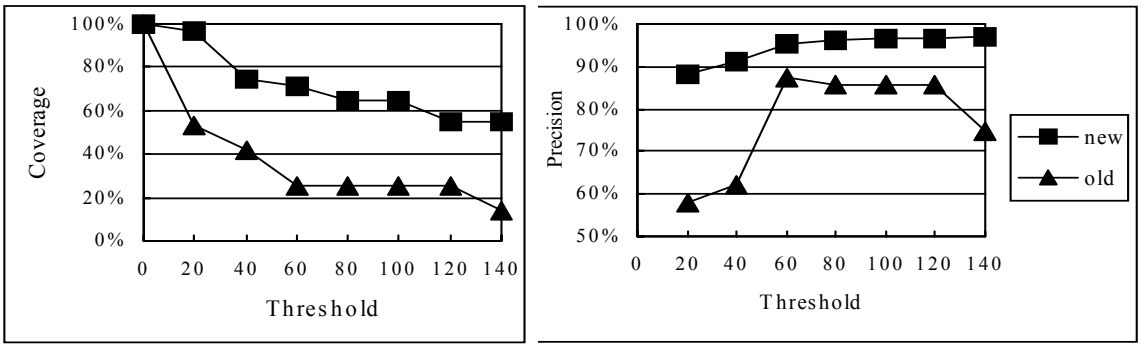
As pointed out in section V, although this approach is motivated by identifying synchronization concern, it can discover other crosscutting concerns as well. For example, in our evaluation, we find fan-in for sk_ is 835: they are related to socket concern; fan-in for FPU_ is 806: they are related to FPU simulation concern. While calculating the precision of the approach, we need to consider all the aspect candidates identified by this approach.

Figure 2(B) shows a comparison of the precision between these two approaches at different thresholds. It shows that the classified fan-in analysis has a higher precision than the original fan-in analysis. The precision of the new approach is still high enough even at very low threshold (from 20 to 40), while fan-in analysis gets a relatively low precision.

At a threshold around 20, the new approach can achieve both high coverage and precision (both around 90%), which suggests that a threshold at 20 might be good enough.

C. Tracing Mining

Considering the poor performance of clone detection to find this crosscutting concern, there is no need to compare the extended classified fan-in analysis approach to clone detection. In this part, we will evaluate the performance of



A. Coverage at different threshold. B. Precision at different threshold.

Figure 2. Performance comparison for classified fan-in analysis.
New: Classified fan-in analysis. Old: Original fan-in analysis.

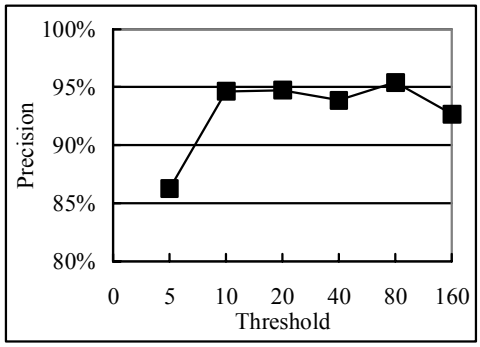


Figure 3. Precision at different threshold for extended classified fan-in analysis

this approach alone. The fan-in for *PT_* prefix is 41. So, when the threshold is over 41, the coverage is 0%, and when it is less than or equal to 41, the coverage is 100%.

This approach is not restricted to tracing aspect. For example, other aspects (such as *X86_, PGDIR_, IPC_,* which are also aspects) can be found by this approach too. So when calculating its precision, the other aspects must be considered.

As shown in Figure 3, when the threshold is around 10, the precision is very high (about 95%), and the coverage for the tracing aspect is 100%. When it is larger than 10, the precision is almost the same, but the approach can find fewer aspect candidates. Thus a threshold of 10 is probably the best for this approach.

VII. CONCLUSION

We present a case study of aspect mining in Linux in this paper. We first identify four important aspects in Linux and apply several existing aspect mining approaches to find these four crosscutting concerns. The results show that these approaches do not perform very well to mine aspects in Linux. We then propose three new aspect mining approaches to improve the efficiency of aspect mining in Linux.

Although more detailed analysis is needed to evaluate each of the techniques proposed, we believe our work has demonstrated some potential towards efficient aspect mining in Linux. Besides mining aspects, it will also be important to explore how to use the aspect candidates obtained to build an aspect-oriented Linux, i.e., to refactor the crosscutting concerns with AOP properly.

VIII. REFERENCES

[1] G. Kiczales, J. Lamping, and A. Mendhekar et al. "Aspect-Oriented Programming." *In proc. European Conference on Object-Oriented Programming*. Finland. 1997.

[2] A. Kellens, and K. Mens, "A Survey of Aspect Mining Tools and Techniques", *INGI Technical Report*, 2005.

[3] T. Tourwe and K. Mens. "Mining aspectual views using formal concept analysis". *In Source Code Analysis and Manipulation Workshop (SCAM)*, 2004.

[4] B. Ganter and R. Wille. "Formal Concept Analysis: Mathematical Foundations". *Spring-Verlag*, 1999.

[5] D. Shepherd, T. Tourwe, and L. Pollock. "Using language clues to discover crosscutting concerns". *In Workshop on the Modeling and Analysis of Concerns*, 2005.

[6] K. Gybels and A. Kellens. "Experiences with identifying aspects in Smalltalk using 'unique methods'". *In Workshop on Linking Aspect Technology and Evolution, AOSD*, 2005.

[7] M. Marin, A. v. Deursen, and L. Moonen. "Identifying aspects using fan-in analysis". *WCRE*, 2004.

[8] D. Shepherd, E. Gibson, and L. Pollock. "Design and evaluation of an automated aspect mining tool". *In International Conference on Software Engineering Research and Practice*, 2004.

[9] M. Bruntink, A. v. Deursen, and R. v. Engelen et al. "An evaluation of clone detection techniques for identifying crosscutting concerns". *ICSM* , 2004.

[10] M. Marin. "Reasoning about assessing and improving the seed quality of a generative aspect mining technique". *AOSD,* 2006.

[11] D. Cao and H. Mei. "Aspect Orientation – A New Approach to Programming". *Computer Science*. 2003.

[12] X. Chen, F. Yang. "Research on Aspect Oriented Operating Systems". *Journal of Software.* 2006.

[13] Y. Coady, G. Kiczales and M. Feeley et al. "Structuring Operating System Aspects". *ICSE, Aspect-Oriented Programming Workshop.* 2001.

[14] Y. Coady, G. Kiczales and M. Feeley et al. "Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System". *In Proc. 9th ACM SIGSOFT, FSE.* 2001.

[15] Y. Coady and G. Kiczales, "Back to the Future: A Retroactive Study of Aspect Evolution in Operation System Code", *In proc. of AOSD,* 2003

[16] D. Mahrenholz, Olaf Spinczyk and Andreas Gal et al. "An Aspect-Oriented Implementation of Interrupt Synchronization in the PURE Operating System Family". *In proc. of ECOOP Workshop on Object Orientation and Operating Systems.* 2002.

[17] M. E. Fiuczynski, R. Grimm and Y. Coady et al. "patch(1) Considered Harmful". *In Proc. Workshop on Hot Topics in Operating Systems*, 2005.

[18] R. A. Åberg, J. L. Lawall and M. Südholt et al. "Evolving an OS kernel using temporal logic and Aspect-oriented programming". *In Proc. AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software.* 2003.

[19] O. Spinczyk and D. Lohmann. "Using AOP to Develop Architectural-Neutral Operating System Components". *In proc. of the 11th workshopon ACM SIGOPS European workshop: beyond the PC.* 2004.

[20] B. M. Cantrill and M. W. Shapiro et al. "Dynamic Instrumentation of Production Systems". *In proc. ACM SIGPLAN conference on Programming language design and implementation*. 2005.

[21] T. Kamiya and S. Kusumoto et al. "CCFinder: A multilinguistic token-based code clone detection system for large scale source code". *IEEE Transactions on Software Engineering*, 28(7):645-670, July 2002.