An Industrial Case Study of Coman's Automated Task Detection Algorithm: What Worked, What Didn't, and Why

Lijie Zou and Michael W. Godfrey
Software Architecture Group (SWAG)
School of Computer Science, University of Waterloo
{lzou, migod}@uwaterloo.ca

Abstract

Programmer interaction histories are event sequences that have been culled from the use of instrumented IDEs, and can be mined to better understand the work habits of developers. A prerequisite for this analysis is an understanding of where the boundaries between distinct tasks lie. Asking developers to indicate manually when they switch tasks is disruptive to their normal work flow, so researchers have sought ways to infer task boundaries automatically based on the content of the interaction histories. Coman previously reported a fully automated algorithm that achieved 80% accuracy in a lab validation study. In this paper, we evaluate the use of Coman's algorithm within an industrial setting. We found two problems: first, a large number of the tasks identified are in fact only sessions of a larger task; second, the demonstrable effects of interruptions are not considered. We argue that the problem of task boundary detection consists of two subproblems: first, detecting task sessions; and second, linking task sessions. Coman's algorithm effectively addresses the first, while the second remains unsolved.

1. Introduction

Recently, interaction history has become a new source of information for understanding how programmers work. Interaction history is a record of a programmer's interactions with an IDE, which may be captured by instrumentation. By mining interaction histories, team coordination can be improved, hidden relations may be detected, usage context may be extracted, and insights into the practice of software development can be obtained [13, 16, 9].

The task is the basic work unit in software development. To better assist programmers, researchers have proposed to support tasks more explicitly in software development tools, such as supporting task management, recommending task-relevant data and sharing task knowledge [8, 12]. However, these techniques rely on the identification of task boundaries, that is, the points in time when a task is started, interrupted, resumed, or finished.

To our knowledge, Coman et al. have proposed the only automated algorithm for detecting task boundaries from interaction histories [1, 2]. In a lab validation study, this algorithm has shown 80% success in identifying the number of tasks and task boundaries. However, industrial software development is quite different from lab setting. Software systems are often much larger, tasks are more complex, and interruptions are common. In this work, we have sought to evaluate how Coman's algorithm may perform within an industrial setting.

We performed an industrial case study involving six professional software developers working on their normal real-world tasks. From over a month's worth of raw data, we examined 12 randomly chosen person-days in detail. We applied Coman's algorithm on the interaction histories captured by the IDE, and compared the output against the "ground truth" as reported by the developers themselves. When we noticed surprising results, we further examined the details of the data to obtain insights into the algorithm and the problem space, and considered possible ways to improve the algorithm.

This paper is organized as follows: we discuss Coman's algorithm, the original lab validation study, and present our research questions in Section 2. We then describe the case study design in Section 3 and the results in Section 4. We discuss related work in Section 5, and threats to validity in Section 6. Finally, we summarize our findings in Section 7.

2. Coman's algorithm and lab study

Coman et al. proposed an algorithm that detects task boundaries automatically based on programmer interaction histories. The main ideas behind the algorithm are:

- Each task has a set of artifacts that are essential for performing the task, namely the *task core*.
- When programmers work on a task, there is a time period when the task core is accessed intensively at approximately the same time.

Based on these assumptions, the algorithm first computes the *time intervals of intensive access* (TIIAs) for each method — these are the time periods with intensive access to the method. Time moments with a large number of TIIAs are then identified as task core locations. Using these task core location as seeds, the TIIAs are grouped into task subsections based on the temporal distances. The resulting task subsections are finally identified as the tasks.

Since we refer to the details of the algorithm in later discussions, we now describe the main steps and parameters of Coman's algorithm:

1. *Compute TIIA(m)*

   Based on interaction histories, the algorithm computes TIIAs of each method based on degree of access (DOA). DOA is defined as $DOA(m,t) = \frac{AT(m,t)}{t-t_0}$, where $AT(m,t)$ is the amount of time that a method is accessed and $(t-t_0)$ is the total interval of time period. By definition, the value of DOA is between 0 and 1.

   A TIIA of a method starts when it is first accessed, and ends when its DOA decreases below a stated threshold, th. The higher the th, the stricter for being considered as "intensive access", therefore the more likely to end an existing TIIA in case of no access. th is a parameter for tuning.

2. *Form TIIA time series*

   Once all of the TIIAs have been computed for all of the methods, the number of TIIAs at each time can be computed. This forms a TIIA time series.

3. *Smooth TIIA time series*

   The TIIA time series is smoothed using weighted central moving average (WCMA). Every data point is computed as an average of the current point and a number of data points on either side, with weights decreasing as the temporal distance increases. Smoothing transforms the TIIA time series into a continuous line so that peaks and valleys can be identified later on.

   The number of data points used in WCMA is a parameter for tuning, namely ta.

4. *Identify task core locations — peaks on the TIIA time series*

   Peaks on the TIIA time series indicate moments when a large number of methods are accessed within a short period. By assumption, these methods are task cores and the peaks are task core locations.

   Only peaks with height greater than a threshold tp are considered as task core locations. The height of a peak is computed as the difference between the peak and the higher of the two adjacent valleys. tp is a parameter for tuning.

5. *Expand task core locations to subsections*

   The task core locations are then expanded by adding all of the TIIAs one by one. Each TIIA is grouped with the task subsection that has the smallest distance to it. The distance of a TIIA and a task subsection is defined as the spanning difference of the task subsection if adding the TIIA.

   Each task subsection is then identified as a task.

The algorithm has three parameters that can be tuned: th for the DOA threshold, ta for the number of data points (or the time) on each side in smoothing, and tp for peak height threshold. Coman et al. state that parameter tuning is important for applying the algorithm, but they do not provide specific techniques or guidelines for doing so.

To test the proposed algorithm, Coman et al. performed a laboratory experiment. Three students were allowed 70 minutes to solve five maintenance tasks for the Paint program (with 9 classes and 503LOC). Both the tasks and the Paint code were developed by other researchers. In this study, the parameter values used were $th = \frac{2}{3} * median(accesses) * \frac{1}{10}$, $ta = 250(4min)$, and $th = 0.2$, where median(accesses) is the median length of the accesses. The definition of th takes the characteristics of programmer accesses into consideration. If a programmer tends to have long accesses, a single lack of access would not make th decreases rapidly and end the current TIIA. Parameter ta was set to 250 , which equals roughly 4 minutes on either side. Peak threshold th was set to 0.2.

In this lab study, the algorithm was able to correctly identify the number of tasks and their approximate time location in 81% of the cases. All of the task subsections that were identified as tasks by the algorithm did indeed correspond to a “ground truth” task.

2.1. Research questions

We know from experience that industrial software development is often quite different from the artificial setting in the lab. In industry, programmers are driven by time pressures, the tasks to be solved are complex, the software systems are often very large, and interruptions are common. All these factors may cause techniques that work well in the lab to behave differently in an industrial setting.

With this in mind, we chose to investigate two research questions:

Q1: How does Coman's algorithm perform in an industry setting?

Q2: If the algorithm performs differently than expected, why?

3. Case study design

To answer our research questions, we performed an industrial case study of detecting task boundaries. The study was performed in Heweisoft, a software company located in Shanghai, China whose main expertise is in building information system for enterprise and government.

Six programmers from the R&D department participated this study; we shall refer to them as P1, P2, etc. All of them were actively involved in the development of an internal platform called H3, a middleware system for distributed enterprise application. The programmers were working on different sub-projects of H3: P1 and P2 were developing H3.DBM, a distributed database engine about 300KLOC, P3 worked for H3.SYS, a system management component about 50KLOC, and P4 was in H3.KNW subproject, a knowledge management system about 100KLOC. At the time of the case study, the H3 platform was being used by two other projects for building business applications and was in active maintenance.

Before the study, each programmer completed a questionnaire about their background and experience. The results are shown in Table 1. At the time the study was performed, P1, P2, P3, and P4 had been on the H3 team for about two years, and all were considered to have good understanding of the whole project. P1, P2 and P4 had switched to their current sub-project six to eight months previously; all had become experienced developers of the sub-project. P5 and P6 were

Table 1. Background of participants

Prog	Proj	Years in Proj	Years in Java	Years in Eclipse
P1	H3.DBM	0.8	4	2
P2	H3.DBM	0.8	5	2
P3	H3.SYS	2	5	2
P4	H3.KNW	0.6	6	2
P5	Learning	0	1	2
P6	Learning	0	2	2

newcomers to both the project and the team, having just joined the company the previous week. During the first period of learning, they were assigned some tasks to become familiar with the software system.

For each programmer, we captured their interaction histories for a month using a special-purpose plug-in tool we created for their development environment, Eclipse. The tool required no input from the user and was invisible to the programmers, although of course they were all aware of its existence and use.

For each programmer, we randomly picked three days before the start of the study as the days for applying the task splitting algorithm. The developers did not know ahead of the time which days had been picked. At the end of each of those days, developers were asked to send email detailing what tasks they had performed that day, and how their time had been broken up into tasks. The developers were asked to do this at the end of these chosen days so that the knowledge of what they had done that day would still be fresh in their memories.

For each of the selected days, we applied the algorithm and compared the results with the number of self-reported tasks, which served as our ground truth. We first ran the algorithm with the same parameter values had been used in the lab study; we refer to these as the *baseline values* for the parameters. We then tuned the parameters to find values that best fit our data. Since Coman et al. provided no guidance on parameter tuning, we did so based on our own understanding of the algorithm: for each combination of parameter values, we applied the algorithm and chose the parameter values that best fit with the self-reported number.

4. Case Study Results

We received a total of 12 email messages from the developers describing their tasks and time spent, although 18 had been expected. Table 2 shows these results, as well as the number of interaction events on those days to give a rough idea of the activity level.

We can see from the table that the number of tasks

Table 2. Basic results

	#Tasks self-rpted/predicted - : no report received			# Interaction Events on the day		
Prog	D1	D2	D3	D1	D2	D3
P1	1/16	2/23	-	7141	22221	-
P2	2/15	3/15	-	10485	10072	-
P3	3/0	2/1	-	1701	1367	-
P4	2/2	-	-	752	-	-
P5	1/1	1/12	1/4	478	6444	1060
P6	-	1/22	2/5	-	8527	1496

everyday is small, ranging from one to three. Most programmers reported only a very rough time segmentation, often one in the morning, and another in the afternoon. We consider this understandable since it is much more difficult to recall the details of time periods than the number of task.

4.1. Q1: How does Coman's algorithm work in an industry setting?

To answer Q1, we applied the algorithm and compare the results with self-reported data, the ground truth. We ran the algorithm under two situations, with baseline parameters and with parameter tuning.

4.1.1 Results with baseline parameters

For each of the 12 days, we apply the task splitting algorithm using the baseline parameters: $th = \frac{2}{3} * median(accesses) * \frac{1}{10}$, $ta = 250$, and $tp = 0.2$. Table 2 shows the results.

We can see that the number of tasks detected fits poorly with the self-reported number overall. The total number of tasks with the baseline parameters was 112, while the total number of the self-reported value was only 21.

The algorithm's accuracy varied over the different days of the study. On two days, P4.D1 and P5.D1, the number of tasks detected was the exactly same as the self-reported. But in half of the 12 total cases, such as P1.D1 and P1.D2, the number of tasks found by the algorithm was much larger than the self-reported number. In the case of P3.D1, no tasks were detected. Further analysis shows that this was because no peak has height greater than the threshold value (the highest peak has height of 0.14).

4.1.2 Results of parameter tuning

We tried to tune the parameters to better fit our data. Since the baseline parameter values worked well in the lab study, we judged it likely that the best parameters for our study would be somewhere close by. Consequently, we performed the parameter tuning around baseline values. For each parameter, we pick a few values below and a few above the baseline values. In more detail, we tune the parameters th, ta, and tp as follows:

- DOA threshold: th

  th is the threshold for the level of DOA that ends a TIIA. It is set to $th = \frac{2}{3} * median(accesses) * \frac{1}{10}$ in the baseline, where $median(accesses)$ is the median duration between accesses. A larger value for th means shorter TIIAs, and thus fewer tasks.

  In our case study, the median(accesses) of the 12 days have total four values, four of 1, four of 2, one of 3, and one of 6. This means the range of th is from 0.067(th=1) to 0.4 (th=6). So following the rationale of "close to the baseline", we choose the following values:

  $th = 0.04, \frac{2}{3} * median(accesses) * \frac{1}{10}, 0.2, 0.4$.

- Smoothing parameter: ta

  ta defines the number of data points on either side involved in smoothing. The larger the ta is, the smoother the time series will be, and thus the peak height will be smaller and fewer tasks will be identified.

  The baseline value of ta is 250, roughly 4 minutes on either side, so we choose following ta values:

  $ta = 180, 250, 300, 420(3min,\ 4min, 5min, 7min)$

- Peak height threshold: tp

  tp is the threshold for peak identification in the time series. The larger tp is, the harder it is for a peak to be considered as a task core location, thus leading to fewer peaks and fewer tasks.

  The baseline value of tp is 0.2, so we try following values:

  $tp = 0.1, 0.2, 0.4, 0.6$

For each combination of parameter values, we ran the task splitting algorithm. The results are shown in Figure 1: the 12 series represent the 12 programmer days, the X-axis represents particular combinations of parameter value, and the Y-axis represents the number of tasks being detected. There are a total of 64 data points on the X-axis, since each of the three parameters has four values. These data points are sorted in the order of th, ta, and tp, starting from $th = 0.04, ta = 180(3min), tp = 0.1$ and ending at $th = 0.4, ta = 420(7min), tp = 0.6$.

As we can see from the figure, some series are quite spiky, such as P1.D1 and P6.D2, while some are

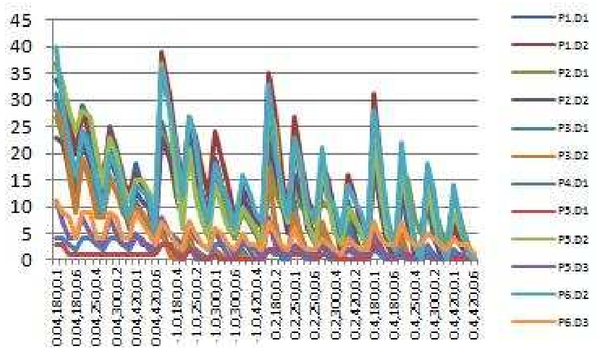

Figure 1. Parameter tuning

smoother, such as P5.D1. We notice a common pattern for a large number of series: a total of 16 spikes on a series, and every 4 form a group; within each group, the height of the four spikes decrease and each spike is sharp; and across the groups, the overall height decreases slightly. This pattern suggests that the effect of parameters has some regularity.

The rapidly decreasing height of each spike indicates that the effect of tp on the algorithm is strong. Increasing tp lifts the threshold of being identified as peaks, and greatly reduces the number of tasks greatly.

Within each group, the decreasing height of the four spikes shows the effect of ta, since the only difference between the four spikes is ta. The bigger ta, the stronger the smoothing, which causes lower peaks and higher valleys. The rapidly lowering spikes within a group suggests that smoothing has a strong effect on the algorithm.

Across the four groups, the change of the overall height is the result of changing th, as the only difference between the groups is th, the threshold of DOA. We found that the four groups look quite similar, with only a slight decrease of the overall height. This suggests that the effect of th on the algorithm is relatively small,

For each parameter combination, we compare the number of tasks being detected with the self-reported number. The parameter sets that have the smallest total difference for all the 12 cases are chosen as the best fits.

We found that the parameters that fit best are $(th = 0.4, ta = 180, tp = 0.6)(51)$, and $(th = 0.4, ta = 420, tp = 0.4)(62)$. In both cases, the total difference of number of tasks identified is 16. The total number of self-reported tasks is 21, so the overall error rate is about 16/21=76%. In both parameter sets, no tasks are detected in half of the 12 programmer days, or that the parameters are over-tuned for these cases. This suggests that parameter tuning for the algorithm may not be as simple as finding a universal set of values. It may depend on factors such as programmer, task, and the software system.

We feel it is important to pay attention to the underlying meaning of the parameter values while tuning the parameters. For example, smoothing means that information is lost. Setting $ta = 600$ means 10 minutes on each side participated in smoothing, which means 20 minutes in total. However, 20 minutes may be long enough to solve a small task. So would such small tasks be washed away in such smoothing? Also for tp, increasing its value means that higher peaks are considered as task cores, which also means that more methods should be accessed intensively in a shorter time period to be identified as a task core. It is unclear that this is a reasonable assumption.

4.1.3 Answer to Q1

From the results presented above, we can see that the performance of the algorithm in our industrial setting is much different from that observed in the lab setting in the original study. With the same parameters as in the lab study, the total number of tasks being detected was about 5 times that of the self-reported number. Results vary with cases: some are the exactly same, some are below, but most are well above the self-reported number. After tuning the parameters, the best cases still have error rate of about 76%. The parameter tuning may be more complicated than finding a universal applicable parameter set.

4.2. Q2: If the algorithm performs differently than expected, why?

Since the results of the Coman algorithm were quite different in our industrial study than the original lab results, we sought to understand why this might be the case by performing a deeper analysis of the data and the assumptions of the algorithm.

4.2.1 Large number of predicted tasks

We wondered why in many cases Coman's algorithm tends to detect many more tasks than the actual number. So we examined the details of the data: the TIIA time series, the peaks, the task core and task subsections. We used P1.D1 under the baseline parameter as the example. On that day P1 reported one task.

Figure 2 shows the TIIA time series of P1.D1 before and after smoothing. 16 peaks on the smoothing series are identified as the task core locations. We note that there is no single peak that is dominant.

We further looked into the task cores, as shown in Figure 3. In this figure, each method that is ever in-

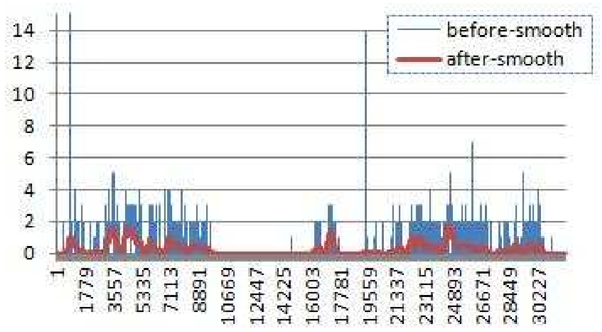

Figure 2. TIIA time series being smoothed

cluded in one task core is plotted on the X-axis. Each row represents a task as detected from the algorithm. A dot at (X, Y) represents that method X is included int the core of task Y. As we can see, the size of the task core is small, about 2 methods on average. There are small overlap between task cores: *e.g.*, the task core of S7 is exactly the same as S14, and is a subset of the task core of S8.

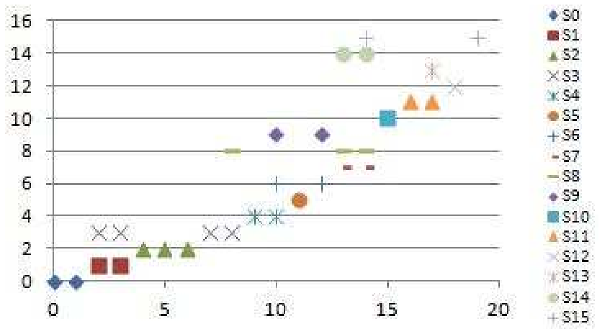

Figure 3. Methods in task cores

Figure 4 shows the artifacts included in each task being detected. This figure is similar to Figure 3 except that X-axis is files and a dot at (X,Y) means that file X is included in the task of Y. We plot files instead of methods since the number of methods is quite large. We can see from the figure that artifacts between tasks overlap greatly. Many consecutive tasks have a large number of common artifacts. For example, all of the artifacts in S9 are contained in S10, same is for S13 and S14.

We used the same methods examining the other days that have a large number of tasks. We observed that the data shows a common pattern: no dominant peak, small size of task cores, small overlap between task cores, and large overlap between task artifacts. This suggests that some tasks being detected by the algorithm in fact belong to one task: the task proceeds through multiple sessions (multiple peaks), with a set of methods that are essential to the session being accessed intensively approximately the same time (peaks); these essential methods may vary over time (task cores), but considering all the artifacts involved in each session, the overlap is large (large overlap between tasks).

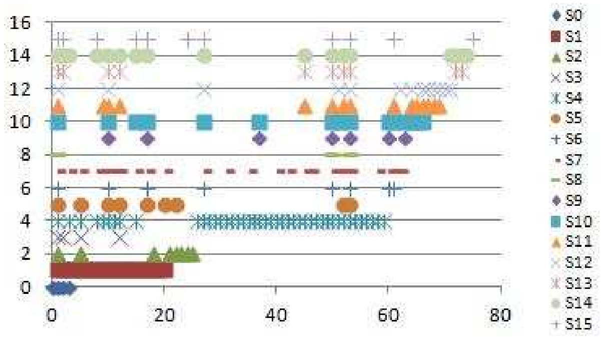

Figure 4. Files in tasks

Having multiple task sessions actually seems natural in an industry setting. For example, a programmer may first analyze the behaviour of a reported bug, then fix the bug, and finally check other code to make sure no new bugs are introduced. In each of these sessions, the goal is different, thus the core artifacts may be different. However, since they belong to the same task, there are links between them, such as common artifacts involved.

Therefore, the underlying assumption of the algorithm that artifacts that are accessed approximately the same time are a task core is problematic in industrial software development. The task cores that are identified by intensive access at the same period are in fact "task session cores". In the lab study, since the software system is small and the task is simple, task session happened to be a task, therefore no problem occurred. But in an industrial setting, this appears to be overly simplistic.

4.2.2 The effect of interruptions

Interruptions to the work flow of a task are not explicitly considered by Coman's algorithm; yet, it seems clear that interruptions can and do have important bearing on real-world task performance.

In the algorithm, a period of "no access" for an method will decrease the DOA and may end the TIIA. When an interruption occurs, the DOA of all the artifacts decreases rapidly, which may result the ending of a large number of TIIAs, forming a gap in the TIIA time series. The gap may result in a valley in the smoothed time series. The more valleys, the more possible peaks, and finally the more tasks. Therefore, in-

Table 3. #Tasks change when INTR shrink

	#Tasks before → #Tasks after		
Prog	D1	D2	D3
P1	16 → 13	23 → 22	-
P2	15 → 14	15 → 15	-
P3	0 → 0	1 → 0	-
P4	2 → 1	-	-
P5	1 → 1	12 → 12	4 → 3
P6	-	22 → 20	5 → 4

Table 4. Task boundaries overlap with INTR

	#Tasks overlap / #Tasks		
Prog	D1	D2	D3
P1	8/16	6/23	-
P2	5/15	6/15	-
P3	-/0	-/1	-
P4	1/2	-	-
P5	-/1	9/12	2/4
P6	-	10/22	4/5

terruptions may affect the number of predicted tasks.

Also in the algorithm, the task boundaries are determined by the overlap of TIIAs. Interruptions may cause large gaps of TIIAs. According to the algorithm, TIIAs before the gap will be grouped into some task before the gap, even though the core of the real task is after the gap. So, interruptions may also affect boundaries of the task.

To assess the effect of interruptions on the number of tasks, we decided to artificially "shrink" the interruptions in the interaction history and compare the number of tasks before and after the shrinking. For example for P1.D1, if we shrink all the interruption longer than five minutes to one minute, then the number of tasks is reduced from 16 to 13. Table 3 shows the difference. We can see that the effect of interruptions does indeed affect the output of the algorithm: in seven out of the 12 cases, the number of tasks decreased.

To see the effect of interruptions on the task boundary, we compared the boundaries of tasks with the interruptions within the interaction history. For example for P1.D1, for the total 16 tasks as detected, 8 of them have a starting time that overlaps with one of the top 20 interruptions.

Table 4 shows the number of overlaps between the task boundary and interruptions for all the 12 days. Overall, one third to half of the task boundaries overlap with the top 20 interruptions. In some cases, such as P5.D2 and P6.D3, the degree of overlapping is large.

We also note that when the parameters are changed, overlapping is still present. For example, Table 5 shows the number of overlaps for P1.D1 under different parameters. We can see that about half of the starting time of the tasks overlap with the interruptions.

Table 5. INTR Overlap when params change

	#Tasks overlap / #Tasks			
P1.D1	$ta = 180$	$ta = 250$	$ta = 300$	$ta = 420$
$tp = 0.2$	8/20	8/16	9/14	5/7
$tp = 0.4$	6/12	6/10	6/8	3/5
$tp = 0.6$	5/8	4/7	3/4	3/5

4.2.3 Answer to Q2

From the above discussions, we can see that there are two problems with Coman's algorithm that may account for its inconsistent performance between an industrial setting and a lab setting:

1) The assumption that the task core is accessed intensively at approximately at the same time is unrealistic. We observed that a task may proceed through multiple sessions. The task cores being identified under such an assumption are in fact "task session cores". In the simple lab setting, a task often only had one task session. But in an industrial setting, this assumption does not hold and resulted in many more tasks being identified.

2) The effects of interruptions on the number of tasks and task boundaries were ignored, yet interruptions are common in real-world development and affect how tasks and task sessions are performed. The lab setting assumed no interruptions, but in an industrial setting with many interruptions the effects were significant.

4.3. Discussion of task boundary detection

In this section, we discuss the general problem of detecting task boundaries: why it is important, what makes it challenging, how to decompose the problem, and how to improve Coman's algorithm.

4.3.1 Background

A programmer's work is driven by tasks, such as fixing bugs or adding new features. However, it should be pointed out that there is no generally accepted categorization of developer task types. In Perry's diary study of 13 industrial developers, 13 types of tasks are found, such as estimation/investigation, code and high-level test [10]. In Erdem's work, a taxonomy for software development tasks was used; it included twenty categories such as synthesis, fault finding, and configuration [3]. In our work here, we consider only software maintenance tasks that include the modification of code. Such tasks are usually categorized into four kinds: adaptive, corrective, perfective, and preventative.

Solving a software maintenance task often involves large amount of effort. Programmers need to go through different stages, such as coming to an understanding of desired behavior, gathering relevant information, and testing hypotheses [3]. At each of these different stages, they usually need to develop goals, hypotheses, and ask questions [15, 3, 14]. Moreover, in a large software system, task-related information is often scattered around in different places [8]. Developers need to navigate and search a lot to find task relevant artifacts.

A task may also be interrupted or switched out, causing extra effort in managing task context. Programmers may interrupt themselves or be interrupted [4]. If the current task changes to a lower priority, it may be switched out in favor of a higher-priority task. Both in cases of interruption and task switching, the context of tasks need to be saved and loaded, resulting in much mental effort. Task switching is considered by programmers as a serious problem in their work [7].

Therefore, it is important to provide explicit task support for software development. Murphy argued that task structure can be used to improve IDEs by providing task-friendly views, to recommend task relevant information and to build a group memory for collaboration [8]. Robillard et al. proposed to support task-aware software development environment based on navigation analysis [12].

A fundamental problem to these task-aware applications is to detect task boundaries, that is when a task is started, interrupted, resumed or finished. Without an understanding of where the boundaries are, supporting tools may unknowingly mix task contexts together and provide inappropriate information at inappropriate time.

4.3.2 The Problem

The problem of task boundary detection is challenging. There are many ways a given developer might choose to start a given task, making it hard to detect the starting point. Moreover, there are various reasons why a task might be interrupted or switched out, making it hard to identify the transition points between tasks.

Comprehension strategy, task type, programmer expertise, and other factors may all affect how a task is started. Some developers may start a task by building and exploring hypotheses (top-down) while some may by reading code (bottom-up) [15]. Some may begin with searching while some with navigation [6, 11]. For corrective tasks, the common first step may be recreating the problem; for perfective tasks, the first step may be comprehending the desired behavior. Even for the same initial step, the methods used may be different: for example, to comprehend the desired behavior, some developers may use debugging while some may prefer searching the documentation [3].

Also, programmer expertise affects how a task may be started. An experienced programmer with a good understanding of the code may go directly into the code locations that need to be changed, while a newcomer may have to explore code for hours just to find the initial starting point.

Various cases of interruption and task switching also make the problem hard to solve. Many events may trigger the transition of tasks: changing of task priority, getting blocked, getting request from colleagues, lunch time/conference, or simply getting tired of current task. These events are often not recorded anywhere, therefore providing little information that a task boundary detection algorithm may rely on.

4.3.3 Problem Decomposition

Interruption and task switching result in fragmented task sessions. Therefore we consider the problem of identifying the task boundaries in fact consists of two sub-problems. 1) detecting task sessions — time periods when programmer is working on a single task continuously, and 2) linking task sessions together, such as grouping task sessions that belong to a same task. Obviously, solving the first sub-problem is the base for solving the second one. But if the second sub-problem is not solved, task sessions of the same task may be incorrectly identified as two different tasks, therefore leaving the larger problem unsolved.

The algorithm proposed by Coman et al. addresses the first sub-problem. Due to the problems in the assumption, the tasks as detected in the algorithm are in fact task sessions. Each task session in the algorithm is a time period having a number of artifacts are accessed intensively at approximately the same time. So in the case of a complex task, multiple task sessions may be involved. Task sessions from this algorithm may also due to interruptions. In a lab setting with small software system, simple tasks and no interruptions, a task session as such equals the whole task. But this is not true in an industrial setting.

4.3.4 How to make it better?

Coman's algorithm may be improved by taking interruptions into consideration. One technique may be making the algorithm be aware of regular interruption time period, such as lunch time. When the DOA computation goes across the regular interruption period, it is handled differently. Another method is to "shrink" the interruption to reduce the gap effect, as we did in the case study. When an interruption occurs, the DOAs of all the methods decrease greatly and may result a gap in the TIIA time series that affects the task boundary detection. So if the interruption time periods are artificially shortened, say from longer than five minutes to one minute, then the decrease of DOA may not be large enough to end all the TIIAs and result a gap. At the same time, since there is still an interruption time period of one minute, the DOA still exhibits a decrement that reflects the effect of the interruption.

There is yet no solution to the second sub-problem of linking task sessions. We think that the key to solving this sub-problem is to find the link between the task sessions. The link may be common artifacts (being viewed or changed), closely related artifacts (such as peer concepts), code clones, or others. As shown in our study, task sessions with a large number of common artifacts that are ever accessed may belong to the same task. It is also likely that task sessions with a small number of artifacts being changed may belong to the same task. In principle, linkage criteria can make use of all kinds of information available in the interaction history, such as artifact, time, action (such as editing, searching, debugging). When task sessions are linked, the purpose of task sessions may be also identified. For example, if a task session with many searches and viewing is followed by a session with a lot of editing, and if the artifacts that are viewed most in the first is changed in the second, then it may be the case that the first task session was to find task relevant information, while the second was to perform the appropriate changes.

We can also improve the algorithm by including new information sources into analysis. Bug tracking system, such as Bugzilla, is commonly used in a development team to track all the reported issues, or tasks. It records various information about each issue, such as priority, time being reported, and time being solved. All of this information may be used together with interaction history to improve the detection of task boundaries. Another information source is the version control system, such as CVS. A version control system records all the changes to a software plus meta-data about the changes, including author identity, date, and possibly a reference to a related bug report. In many development teams, it is common to commit changes right after the task is solved, and input task information (or the bug ID) when committing changes. Therefore, depending on the tools used and the known practices of the development team, the rough time of a task may be recorded in the version control system and could be used in identifying task boundaries.

4.4. Summary

Our study has shown that Coman's task splitting algorithm tends to detect more tasks than self-reported. Detailed data examination suggests that the underlying assumption about task core is unreasonable and the effects of interruption are not considered. Further analysis of the general problem of task boundary detection shows that the problem consists of two sub-problems. Coman's algorithm addresses the first sub-problem, while the second sub-problem remains unsolved.

5. Related Work

5.1. Mining Interaction history

Interaction history contains rich information about how program artifacts are accessed during software maintenance. Recent research suggests that it is a promising source for better understanding software development. For example, Schneider et al. proposed to study programmer interactions to improve team coordination in distributed projects [13]. Parnin et al. developed a technique to extract usage context by mining interaction histories [9]. Zou and Godfrey used interaction histories to detect couplings [16].

Understanding how programmers structure their tasks is a prerequisite of understanding software development. Coman et al. propose the only algorithm for detecting task boundaries automatically by mining interaction histories [1, 2]. In our study, we apply this algorithm in an industry setting.

5.2. Task-aware Software Development

Tasks are the work units of software development. Researchers have proposed to support tasks explicitly to assist programmers work. Murphy suggested to improve IDEs and collaboration based on task structure — the subset of program artifacts and relations that are relevant to a task[8]. Kersten et al. used task context to recommend task relevant artifacts [5]. Robillard et al. proposed to support task-aware software development environment based on navigation analysis [12].

As Robillard and Murphy pointed out, detecting task boundaries is a challenge for building task-friendly tools [12]. This is what Coman's algorithm tries to solve.

6. Threats to Validity

In our case study, the information of tasks are collected from programmers' self-reports. This assumes that programmers know what a task is, they can recall what they did, and they are willing to report it to us. We consider this not a big problem since we only ask programmers to recall the task on that day, which would not cause too much effort. Also, we ignore days that programmers did not reply to ensure that for the data they do report, the data is accurate.

As with all the case studies, this research has external validity threat. We chose this company only because it was available to us. The programmers participated the study because they were willing to do so. More case studies should be performed to answer our research questions in a more generalized basis.

7. Conclusion and future work

In this paper, we performed an industrial case study of Coman's automatic algorithm for detecting task boundaries from interaction histories. Based on the data from six programmers working for 12 days, we found that the algorithm performed quite differently comparing with the original lab validation study: many more tasks than self-reported were detected, and the best results after parameter tuning still had about 70% of error. Further analysis of the data led us discover two problems within Coman's algorithm: first, the underlying assumption about task core may not be reasonable in an industrial setting; and second, the demonstrable effects of interruptions are ignored.

We discussed the general problem of task boundary detection and argued that the problem consists of two sub-problems: identifying task sessions and linking task sessions. Coman's algorithm in fact addresses the first sub-problem. The second sub-problem remains unsolved. We discuss possible techniques that may be used to improve Coman's algorithm and to address the second sub-problem.

References

[1] I. D. Coman. An analysis of developers' tasks using low-level, automatically collected data. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering(ESEC/FSE)*, pages 579–582, Sept. 2007.

[2] I. D. Coman and A. Sillitti. Automated identification of tasks in development sessions. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, pages 212–217, June 2008.

[3] A. Erdem, W. L. Johnson, and S. Marsella. User and task tailored software explanations, 1999.

[4] H. C. Gall and M. Lanza. "constant, constant, multi-tasking craziness": Managing multiple working spheres. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 113–120, 2004.

[5] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, pages 159–168, July 2005.

[6] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, pages 971–987, 2006.

[7] T. D. Latoza, G. Veolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06, Proceedings of the 28th International Conference on Software Engineering*, pages 492–501, 2006.

[8] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Cubranic. The emergent structure of development tasks. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, July 2005.

[9] C. Parnin and C. Görg. Building usage contexts during program comprehension. In *ICPC '06: Proceedings of the 14th Interaction Conference on Program Comprehension(ICPC 2006)*, 2006.

[10] D. E. Perry, N. A. Staudnmayer, and L. G. Votta. People, organizations, and process improvement. *IEEE Software*, 11(4):36–45, 1994.

[11] M. P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering(ESEC/FSE)*, pages 11–20, Sept. 2005.

[12] M. P. Robillard and G. C. Murphy. Program navigation analysis to support task-aware software development environments. In *Proceedings of the ICSE Workshop on Directions in Software Engineering Environments*, pages 83–88, 2004.

[13] K. A. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a software developer's local interaction history. In *MSR '04: Proceedings of the 2004 international workshop on Mining software repositories*, 2004.

[14] J. Sillito, G. C. Murphy, and K. D. Volder. Questions programmers ask during software evolution tasks. In *SIGSOFT '06/FSE-14: Proceedings of the 14th SIGSOFT international symposium on Foudations of software engineering*, pages 1–11, 2006.

[15] M.-A. Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14:187–208, 2006.

[16] L. Zou, M. W. Godfrey, and A. E. Hassan. Detecting interaction couplings from task interaction histories. In *ICPC '07: Proceedings of the 15th Interaction Conference on Program Comprehension(ICPC 2007)*, 2007.