# Fitting a workflow model to captured development data

Min Zhang
*Garmin International*
min.zhang@garmin.com

Lorin Hochstein
*USC Information Sciences Institute*
lorin@east.isi.edu

## Abstract

*In this paper, we introduce a semi-automated process called software engineering workflow analysis (SEWA) for developing heuristics that analyze captured data to identify where programmers spend their time. To evaluate our process, we ran two case studies in the domain of high-performance computing to generate programmer workflow models for small problems, cross-checking our results against direct observations.*

## 1. Introduction

As software engineering researchers, we are often interested in building and evaluating technologies to improve programmer productivity: to allow programmers to do more work in less time.

In our research, we are interested in estimating the impact of new software productivity tools (e.g., languages, libraries, IDEs) in the domain of high-performance computing (HPC), which involves highly parallel computing systems. Productivity evaluation of such tools is important because HPC systems are expensive, and procurement agents want evidence to support claims that system A is more productive than system B.

In particular, we are interested in the domain of computational science: software that simulates the behavior of complex phenomena, such as the behavior of the Earth's climate or of the inside of a star. Such software can take years to develop before being suitable for production use, and may be maintained for decades.

We can run controlled experiments using small problems to compare technologies, but establishing the external validity of such studies is a challenge. If we had more detailed information about the scientific software process[1], we could use the information from controlled experiments to estimate the impact on productivity. For example, if we knew that projects spent 60% of their time debugging, and we measured a 30% time reduction in a debugging task in an experiment using a new tool, we could estimate an expected 18% reduction in total development time. Alternately, if the distribution of time spent in different programming activities in controlled experiments is similar to the distribution in real projects, we would have more confidence in the external validity of the results.

Unfortunately, we found no quantitative computational science process data in the literature. Our goal, therefore, was to try to identify HPC software development process (specifically, time distribution of programming activities) from data we could capture non-intrusively. We knew of no such tools that had rules for identifying the kinds of HPC-specific activities we were interested in (e.g., parallelizing a sequential program, performance testing).

In this paper, we discuss a methodology we developed for determining heuristics for identifying programmer workflow from low-level captured data, for HPC. To evaluate the accuracy of our heuristics, we conducted two small case studies where we had some additional data from an observer for validation.

## 2. Related work

Several studies have been done to that directly measure the activities programmers engage in and how much time they spend in these activities. For example, Perry, Staudenmayer and Votta conducted such *time and motion* studies of programmers in a telecommunications company [14]. They used programmers' journals, time diaries and direct observation to study programmer behavior. Ko et al. used direct observation to characterize the role that programmer information needs play in programmer decision-making [13]. They identified find twenty-one

[1] We use "process" and "workflow" interchangeably

activities of interest, such as writing code and submitting a change.

While direct observation can provide high-quality results, these studies were not feasible in our case because of resources constraints. As an alternative, we sought to identify work processes through analysis of logs of captured event data, a technique referred to as *process mining* [17].

Researchers have developed several software-engineering-specific process mining tools. Balboa [2] was an early system developed by Cook and Wolf, which was used to detect and characterize the difference between a software process model and the software process practice [3]. More recently, Johnson et al.'s Hackystat system [11] is a non-invasive metrics collection system that was used as the basis for a technique called Software Development Stream Analysis (SDSA) [10] to analyze automatically captured data. Johnson and Kou applied SDSA to identify whether a programmer was following test-driven development. PROM [18] is another non-invasive metrics collection framework that has been applied to try to identify programmer activities [19]. While useful to us as building blocks, none of these approaches focused on the activities we were interested in for the HPC domain.

## 3. SEWA Methodology

Sanderson and Fisher introduced Exploratory Sequential Data Analysis (ESDA) for analyzing observational data in human-computer interaction studies [15]. Tasks that users perform are inherently sequential, involving a series of steps, such as an office worker composing and sending an email. Researchers can now collect large volumes of data in studies, using methods such as instrumenting the computer system used in the study, and through audio/video recordings. However, this results in a large body of captured data that can be difficult to make sense of when initially being explored by the researcher who is not necessarily looking to test a specific hypothesis. Sanderson and Fisher outlined a set of eight generic data transformations ("the eight Cs") that can be applied to sequential data to bring out the elements that are relevant to the research questions at hands.

Ritter and Larkin proposed a methodology called traced-based protocol analysis as a type of exploratory sequential data analysis for developing a process model to summarize sequential data (TBPA) [12]. The term "protocol analysis" refers to a method that involves observing the behavior of human subjects as they perform tasks, where the subject is encouraged to describe out loud (talk-aloud) what they are doing [4].

```
06/04/08 09:18:56, [editor]  emacs
...
06/04/08 09:45:50,[shell] cc -o buffon serial.c
06/04/08 09:45:50, [compiler] compile success
06/04/08 09:46:10,[shell] ls
06/04/08 09:46:12,[shell]./buffon
06/04/08 09:46:24,[editor]"Save File" (serial.c)
06/04/08 09:46:29,[editor]"State Change" (serial.c)
06/04/08 09:46:31,[shell] ./buffon 1 2 3 4
06/04/08 09:46:40,[editor],"Save File" (serial.c)
06/04/08 09:47:29,[editor],"State Change" (serial.c)
...
06/04/08 09:55:04,[shell], cc -o buffon serial.c
06/04/08 09:55:04,[compiler] compile success
```

**Figure 1. Captured data**

In trace-based protocol analysis, instead of using verbal reports, the researcher uses traces of data captured automatically from instrumentation on the user's computer.

We developed the SEWA methodology by applying the concepts of ESDA and TBPA to analyze software engineering data that can be captured automatically from a programmer's environment in order to build a model of where they spend their time at a very fine resolution (on the order of minutes). In this section, we describe how the eight Cs of ESDA (coding, chunking, commenting, comparing, converting, computing, connecting, constraining) can be used to build and evaluate a model of programmer activity from software engineering data.

To illustrate these concepts in a concrete way, we explain them based on examples from our case studies, described in more detail in Section 5. In these studies, a participant was asked to solve a parallel programming problem on a high-performance computing (HPC) system. We instrumented a participant using Hackystat [11] and Umdinst [7], capturing data such as editor events, compiler invocations (including source code) and shell commands. A sample of captured data is shown in Figure 1. We use specific examples from our own studies to illustrate the application of the techniques. The specific strategies we discuss were derived from studies involving a lone programmer working on a small problem in HPC. While these particular strategies are unlikely to generalize, the general approach to model-building should apply to any software domain, provided that the researcher can capture sufficiently rich data from the environment.

### 3.1. Coding

*Coding* is the process of assigning a label from a finite set to a unit of analysis. For example, when analyzing shell commands, we coded the invocation of a text editor (e.g., Emacs, vi, pico) as an *edit* event. We coded the invocation of a command to compile a program (e.g., cc, make) as a *build* event. When analyzing compile invocations, we coded compile as *compile success* event or *compile error* event. Coding reduces the data into simpler categories when possible. Our codes fell into two categories: the higher-level activities of interest (described in Section 3.9), and lower-level activities that were easy to identify from the captured data in a first pass: compile success, compile error, viewing, filecommand, editing, building, running, other, unknown.

### 3.2. Chunking

*Chunking* is the process of breaking up the data into related segments. One basic chunk represents a single type of programmer activity (e.g., debugging). Our general chunking strategy was to use compiles and program executions as boundaries to defined chunks. We reasoned that at compilation or execution time, the programmer would wait and see what would happen, and then take additional action based on the output. A basic chunk is defined as the interval between successive compile/run events.

We first separated the sequence of captured data into logical chunks by basic chunk, where we think the programmer is engaged in only one activity of interest. Chunking can be applied hierarchically, so that we merge adjacent basic chunks into larger chunks.

### 3.3. Commenting

*Commenting* is the process of attaching free-form annotations to units of analysis. To each chunk, we attach a code that describes a high-level activity, and a comment, which consists of an activity code and an annotation to justify why we have chosen to attach a particular activity to a chunk. These comments provide us with traceability, indicating why a particular chunk was coded.

### 3.4. Comparing

*Comparing* refers to the process of examining pairs of data elements to identify similarities or differences. We primarily used comparisons in looking at diffs from code snapshots of successive compiles to identify what programmers were doing.

### 3.5. Converting

*Converting* is the process of transforming the data into a different form. We use converting to transform the lower level codes described in Section 3.1 (e.g., filecommand, editing) into a higher-level programmer activity sequence (e.g. parallel coding, debugging). The set of high-level programmer activities will be very dependent on the type of software being developed and where in the lifecycle the software development process is. Further details of our high-level activities can be found in Section 3.9.

We also use conversions to transform data into more convenient formats for analysis. For example, the data collection tools that we use capture data in XML format, which we then transform into CSV files so that they can be viewed in spreadsheet format (see Section 4.3 for details of this process). We use ActivityGraph (see Section 3.10) to convert the data into a visual format.

### 3.6. Computing

*Computing* often refers to statistical analysis of the data, although it can refer to any type of algorithmic processing that does not require human intervention. We used computing primarily to calculate the distribution of programmer effort across the different activities.

### 3.7. Connecting

Connecting is identifying relations among the data elements. There are many connections among our sequence data that become apparent when the data is inspected visually by a researcher.

One example can be seen in Figure 1. We suspect that the shell command "`./buffon`" is an incorrect use of the buffon program built by the user. We have evidence for this because the following command "`./buffon 1 2 3 4`" there is command-line argument, which is missing from the first invocation. We can confirm this by examining the source code from that time period and verifying that the program does require command-line arguments.

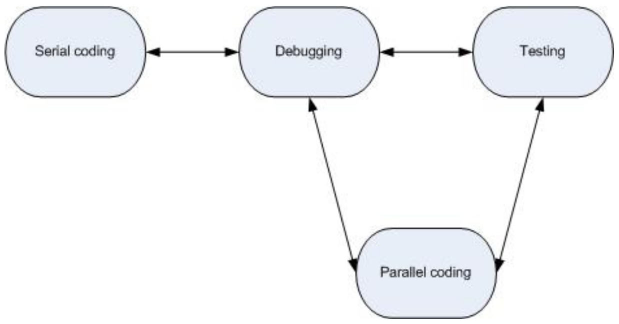

**Figure 2. Preliminary model**

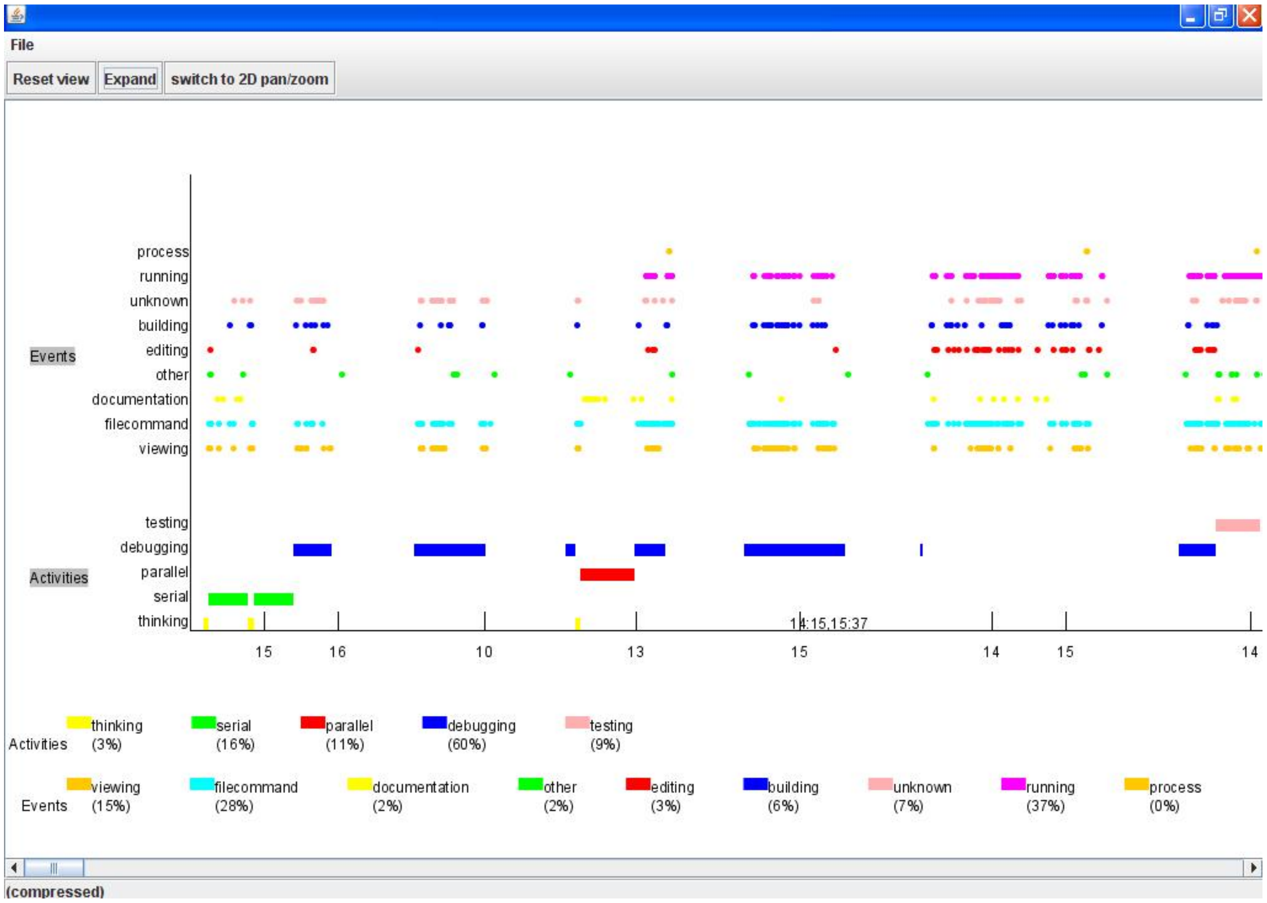

**Figure 3. ActivityGraph**

## 3.8. Constraining

*Constraining* is the process of filtering data that is not relevant to the current analysis. For example, when doing our analysis we were typically not interested in the details of the low-level editor events, so we initially filter those out to focus on the other captured events.

## 3.9. Constructing and fitting a model

The ultimate goal of this activity is to develop an accurate model of programmer activity, at a level of abstraction that addresses the research questions. For our studies, we chose a finite-state model because of its simplicity, depicted in Figure 2.

We began by assuming four categories of workflow activities in the single programmer high-performance-computing work process: *serial coding, parallel coding, testing*, and *debugging*. We defined serial coding as the activity of adding functions and modifying functions to a sequential program. We defined parallel coding as the coding activity to add or modify functions in a program that contain parallelism.

Once we chose our initial model, we applied the eight Cs to analyze the captured data, compared the data to the model, refined the model to better reflect the data, and iterated.

## 3.10 Tool support: ActivityGraph

We developed a tool called ActivityGraph for visualizing temporal, categorical data. It supports ESDA by providing a visual representation of the data to complement a spreadsheet representation. With the visualization activity tool can help researchers know where programmers spend their effort and understand programmer workflow.

ActivityGraph is an open source, zoomable visualization tool for viewing time-series data. It is written in Java and uses the Piccolo toolkit [1] to

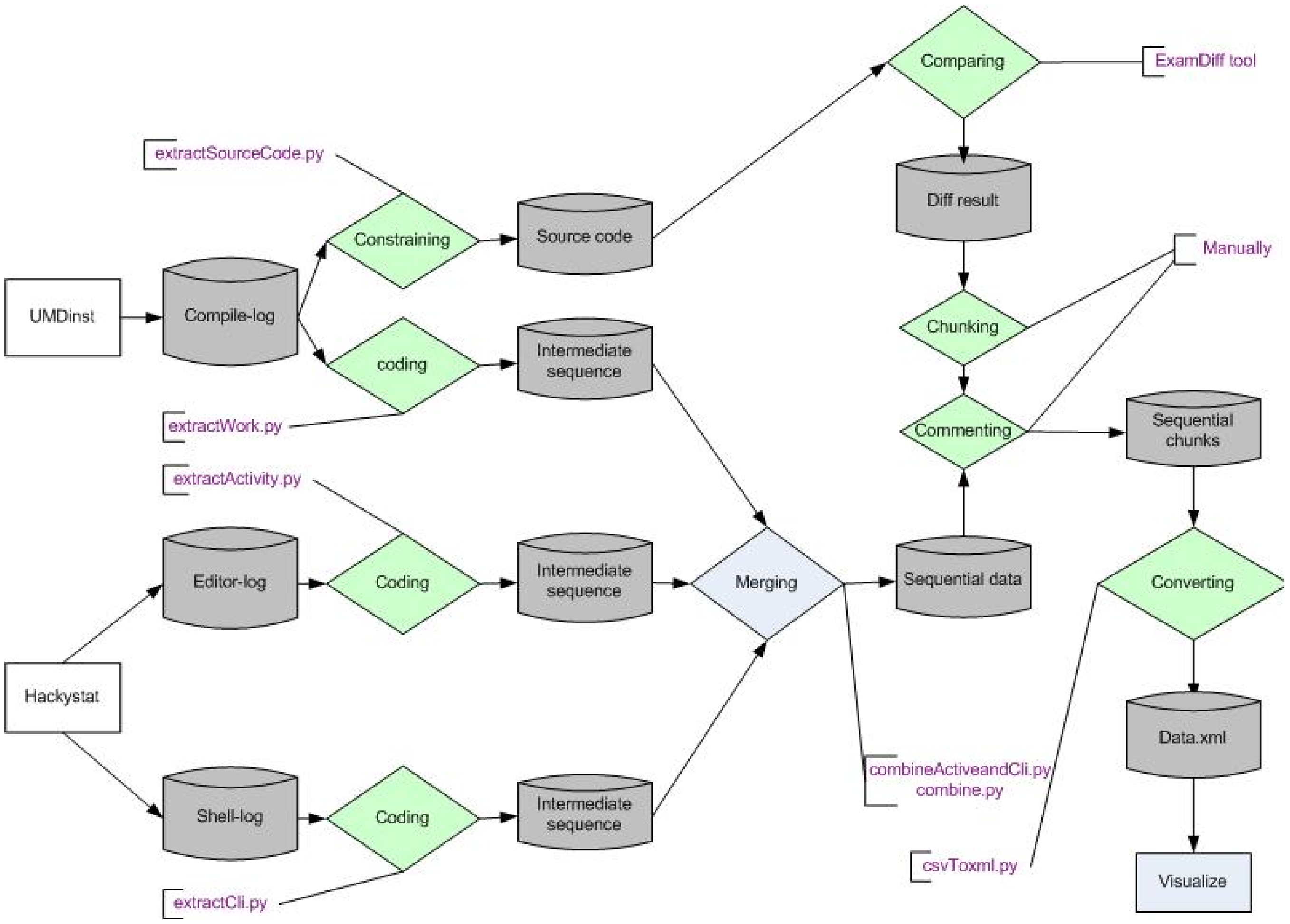

**Figure 4. SEWA process**

support zooming, and is available for download[2]. It is designed to show both low-level events and high-level activities. It also can compute the distribution of the effort across categories. For example, the tool shows a graph and computes the programmer effort (activities) and counts of captured data (events) as shown in Figure 3.

## 4. Applying SEWA to HPC

We tailored our process specifically to HPC, by analyzing data from two studies where the participants wrote programs to run on HPC machines.

### 4.1 Background

To develop and evaluate the work process, we used data from observational studies previously conducted at the University of Maryland. In these studies, an observer kept notes as subjects performed programming tasks. These studies were originally conducted to evaluate the accuracy of an effort estimation algorithm [8]. In all of these studies, the subjects solved small parallel programming problems on a Linux cluster using the MPI library [9]. High-performance computers are often limited a resource and are shared among several users. Therefore, they often use batch-scheduling. A user submits a job to a queue, and when the system has sufficient resources, it launches the job. Hackstat and Umdinst were installed on subjects' computers, which captured compile information, edit events, and shell commands. Because we have access to information from an observer, we used these studies to help understand how well our process works.

[2] http://code.google.com/p/activitygraph

## 4.2 Available Data

These studies used three separate data sources. Compile-log files are produced by Umdinst and include file name, compile time, and source code snapshot. Editor-log files are generated by Hackystat plugins to editors and capture information about editing events. They include information such as edit time, editor name, event type, file name, and file address. Shell-log files are generated by Hackystat plugins to Unix command shells (e.g. bash, tcsh). For each shell command, Hackystat logs the timestamp, command, parameters, and machine name information in it.

## 4.3 Details of the Process Application

Figure 4 shows an overview of the entire process, indicating where the eight Cs are used (not shown are *computing*, which is done by ActivityGraph in the "Visualize" box, and *connecting*, which is a manual process along with chunking and commenting). Files are depicted as cylinders and processing steps are depicted as diamonds. Where a processing step is automatic, the name of the script is shown.

**4.3.1 Standardizing the data.** We began our analysis by examining all of the different kinds of shell commands that appeared in the logfiles. We then devised categories for these commands, and iterated. Our initial coding scheme is shown in Table 1.

**Table 1. Shell command coding scheme**

| Event | Commands |
|---|---|
| *filecommand* | `ls, pwd, rm, cp` |
| *building* | `cc, make` |
| *process* | `ps, top` |
| *documentation* | `man` |
| *running* | `qsub,qstat,qdel` |
| *viewing* | `more,sort,diff,tail` |
| *editing* | `emacs,xemacs,vi` |
| *other* | `xterm,exit,time,xset,gnome-terimnal,date` |
| *unknown* | (all other commands) |

In addition, all of the events captured from the instrumented editor were coded as *editing* events.

The process for standardizing the data is:

1. From shell instrumentation files all the shell commands are coded with event categories.
2. All the event information together with timestamp, commands, argument are extracted and written in a spreadsheet in a time sorted sequence.
3. From Activity files, all the edit event, timestamp, editor name, edit type and source file directory, are put in the spreadsheet sequentially.
4. From work files, all the source codes are extracted into files and named by timestamp; all the compiling situations (success or failed), timestamp are filtered and written into the spreadsheet.
5. According to timestamp, we combine all the information extracted before into one sorted sequence, and write the sequence into a spreadsheet file.

**4.5.2 Chunking the data.** As described in Section 3.2, the data is divided into many chunks using compiles and program executions to delineate chunk boundaries. Each chunk contains data that has been coded into nine high-level events and compiling events. Since each compile is associated with source code, each chunk relates to a version of source code that does not change within the chunk and often changes across chunks. (When chunks are separated by runs instead of compiles, there is no change to the source code across chunks).

In our study, each chunk is typically associated with a changed source file. We use a graphical *diff* tool for visualizing changes to check what kind of changes the programmer made in this chunk. We developed heuristics based on our examination of the diffs.

**4.3.3 Refining the initial model**. As mentioned in Section 3.9, we began by positing a development model with the following activities: serial coding, parallel coding, testing, debugging.

We found it is difficult to distinguish whether the execution of a program should be classified as debugging or testing. For example, when the programmer added a print statement "`printf("count=%d\n", count)`", we assumed this was debugging activity to print out the count result. Following this activity the programmer ran the program with shell command "`./buffon 1 2 2 1000000`" and we found it hard to judge whether this action should be classified as "debugging" or "testing". We resolved this by combining "debugging" and "testing" into a single activity.

After we went through one SEWA iteration, we discovered two activities we could recognize that were of interest to us that were not in our original model: *Fix compile error* and *performance testing*.

*Fix compile error* is the activity that is easy to identify by observing a successful compilation that follows a failed compilation. *Performance testing* is the activity to measure how long it takes the program to run, especially on multiple processors. In HPC, performance testing can be an important activity, which we hadn't thought of initially but discovered in our analysis.

To code the chunks into one of the five programmer activities, we apply heuristics that we developed by inspecting the data. We also recorded comments that explain why a particular chunk was coded a certain way.

**4.3.4 Heuristics for identifying activities.** Based on our analysis of the data, we have developed the following twelve heuristics. Because they are context-dependent (e.g. see #3 vs. #10), they rely on human judgment and cannot be automated in a simple manner.

Heuristic 1: The first compiling action is coded as serial coding.

Reasoning: We assume that when a programmer starts to solve a new problem, she will begin with serial coding.

Heuristic 2: After a successful compile, if the programmer adds more code such as new functions, or update original serial code, it is coded as serial coding.

Reasoning: This is based on the analyst's judgment about the types of changes being made in the source code.

Heuristic 3: Running the program is coded as debugging/testing.

Reasoning: We assume that a programmer runs the program to test for correctness, which is either part of testing (if the programmer thinks the code is correct) or debugging (if the programmer thinks the code is incorrect).

Heuristic 4: All "qstat, qsub" which are not in performance testing situation will be commented as debugging/testing.

Reasoning: Same reasoning as heuristic 3. Qsub is the command to submit a batch job to the batch scheduler. Qstat is the command to print the job's status (e.g., waiting to run, running). If we have reason to believe the programmer is doing performance testing (see heuristics 10,12), then we would not classify as debugging/testing.

Heuristic 5: If the diff between successive compilations includes print information or "DEBUG" strings, it is probably debugging/testing.

Reasoning: Adding print statements to view the internal system state, or using the term DEBUG in the code, are both clues that the programmer is engaged in debugging activities.

Heuristic 6: A series of failed compilations followed by a successful compilation is coded as fix compile error.

Reasoning: Failed compile should typically be recognized immediately by programmer who will then work to fix it.

Heuristic 7: If MPI-related code appears in the diff result of two source files, we will comment this parallel coding.

Reasoning: Calls to the MPI library are how programmers implement parallelism with the MPI programming model.

Heuristic 8: If the file name suggests a parallel program (e.g. parallel.c), it is probably parallel coding.

Reasoning: Programmers sometimes encode semantic information about the program in the filename.

Heuristic 9: If the programmer uses the Unix command "time" or other tool to record the time, code as performance testing.

Reasoning: Time information is typically used for performance analysis.

Heuristic 10: If the programmer continues running the program on different number of processors without modifying the source code, it is probably performance testing to measure how well the program scales

Reasoning: If the program is not being modified, the programmer probably believes the code to be correct. To measure the scalability of the code, which is a performance test, it must be run on different number of processors.

Heuristic 11: If the error log is viewed after the program completes, and then there is a source code modification, we code it as debugging/testing.

Reasoning: The PBS batch queuing system generates files for standard output and standard error. The standard error file also contains PBS-specific errors. If the programmer looks at the error file, it seems likely that a problem has occurred with the executing and the programmer is trying to identify what went wrong.

Heuristic 12: If there are optimization-related compiler flags used, it is probably performance testing.

Reasoning: Since optimization flags improve the performance of the code, it seems likely that a programmer would not experiment with different performance optimizations unless they were doing performance analysis.

We use these rules to comment and label each chunk with a program activity. At the same time we attach an annotation as evidence to each comment.

## 4.4 Process model

In our preliminary workflow model we only use the four categories of workflow activities as mentioned in Section 3.9. After we examined the data, we modified the model accordingly: adding fix compile error and performance testing and combining debugging and testing. These are discussed in the case studies

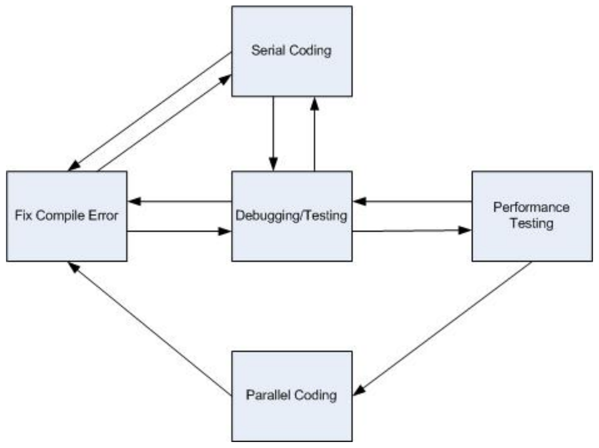

**Figure 5. Workflow model #1**

## 5. Case studies

We needed to know how effective these heuristics were in accurately identifying the software development activities of interest. To validate the accuracy of approach, we took advantage of data from observational studies that had previously been run at the University of Maryland. These studies had an observer who kept a log of the programmer's activities. This log provided us with a benchmark which we could use to cross-check against the results of our process.

### 5.1 Study #1

In study #1, the subject solved the Buffon-Laplace needle problem [16], which is a method for estimating π using Monte Carlo simulation. This simple problem is a good candidate for high-performance computing because the simulations are independent and can be run in parallel.

This was a brief problem, requiring only about two hours of time for the subject to complete the task. After applying SEWA on the data we obtained the results shown in Table 2.

**Table 2. Activity #1**

| Activity | Start time | End time |
|---|---|---|
| Serial coding | 09:18am | 09:45am |
| Debugging/Testing | 09:45am | 09:46am |
| Serial Coding | 09:46am | 09:55am |
| Debugging/Testing | 09:55am | 09:55am |
| Serial Coding | 09:55am | 09:57am |
| Fix Compile Error | 09:57am | 09:59am |
| Serial Coding | 09:59am | 10:04am |
| Fix Compile Error | 10:04am | 10:05am |
| Debugging/Testing | 10:05am | 10:12am |
| Performance testing | 10:12am | 10:12am |
| Parallel Coding | 10:12am | 10:20am |
| Fix Compile Error | 10:20am | 10:21am |
| Debugging/Testing | 10:21am | 10:45am |
| Fix Compile Error | 10:45am | 10:46am |
| Debugging/Testing | 10:46am | 11:05am |
| Performance testing | 11:05am | 11:19am |
| Debugging/Testing | 11:19am | 11:26am |
| Fix Compile Error | 11:26am | 11:26am |
| Debugging/Testing | 11:26am | 11:27am |
| Performance testing | 11:27am | 11:34am |

Notice how quickly the subject moves between states, on the order of minutes. Table 3 summarizes the results.

**Table 3. Activity effort distribution and event effort distribution #1**

| Activity | | Event | |
|---|---|---|---|
| Serial coding | 32% | Compile success | 4% |
| Parallel coding | 6% | Compile fail | 1% |
| Performance testing | 16% | viewing | 0% |
| Debugging/testing | 45% | filecommand | 8% |
| Fix compile error | 1% | editing | 44% |
| | | building | 3% |
| | | unknown | 0% |
| | | other | 1% |
| | | running | 38% |

From the analysis we obtain the programmer's workflow model. Note that in the data above (see Figure 5), there was no observed transition from parallel coding to debugging/testing. However, because this problem is so small, and there is only a single instance of parallel coding, we do not believe this is a realistic model of development for larger problems.

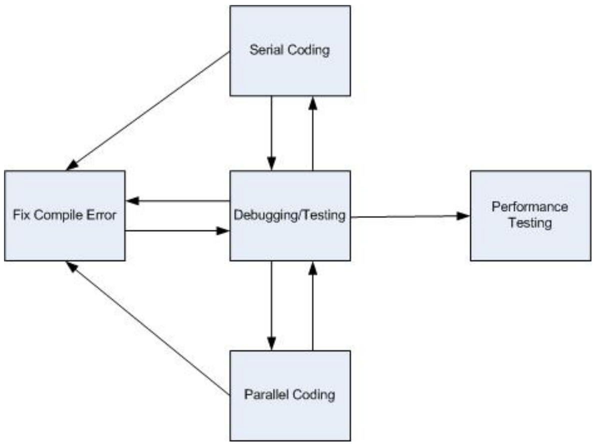

**Figure 6. Workflow model #2**

In this study, we identified five programmer activities in our model: serial coding, parallel coding, debugging/testing, performance testing, fix compile error.

The observer identified nine programmer activities: thinking/problem, thinking/solution, experimenting functionality, parallelizing, tuning, debugging, testing, other. Table 4 shows how we matched up our categories with the observers, and the quality of the match. We associated our serial coding effort with the total effort of thinking/problem, thinking/solution and functionality of the observer. The 2% difference is because there is a small interval of thinking/solution in the observer's log that we cannot see in our data.

**Table 4. Activity relationship #1**

| Observer | | SEWA process | | Diff. |
|---|---|---|---|---|
| Thinking/problem<br>Thinking/solution<br>Functionality | 34% | Serial coding | 32% | 2% |
| Parallelizing | 25% | Parallel coding | 6% | 19% |
| Debugging<br>Testing<br>Tuning | 37% | Debugging/testing<br>Fix compile error<br>Performance testing | 62% | 25% |
| Other | 3% | | 0% | 3% |

A large difference in effort exists for the parallel coding between the study and the observer. The study has 16% parallel coding and the observer has 25% of it. That is because in the observer treats the debugging among the parallel period as parallelizing, which decreases the effort of debugging while increases the effort of parallelizing. For the left debugging/testing activity, fix compile error activity and performance testing activity, they consume 62% of all the effort. The corresponding observer activities: debugging, testing and tuning consume 37% of the whole effort.

## 5.2 Study #2

The problem for the second study is the game of life [6]. This problem required about nine hours of time for the subject to complete, which took place in six development sessions, each on a different day. Table 5 shows the results of our analysis.

**Table 5. Activity #2**

| Activity | Start time | End time |
|---|---|---|
| Serial Coding | 2:12pm | 2:30pm |
| Fix Compile Error | 2:30pm | 2:31pm |
| Debugging/Testing | 2:31pm | 2:39pm |
| Fix Compile Error | 2:39pm | 2:40pm |
| Debugging/Testing | 2:40pm | 2:40pm |
| Serial Coding | 2:40pm | 2:46pm |
| Debugging/Testing | 2:46pm | 2:47pm |
| Serial Coding | 2:47pm | 3:23pm |
| Fix Compile Error | 3:23pm | 3:23pm |
| Debugging/Testing | 3:23pm | 3:32pm |
| Serial Coding | 3:32pm | 3:35pm |
| Debugging/Testing | 3:35pm | 4:01pm |
| | | |
| Debugging/Testing | 9:03am | 10:05am |
| | | |
| Debugging/Testing | 12:04pm | 12:10pm |
| Parallel Coding | 12:10pm | 1:00pm |
| Debugging/Testing | 1:00pm | 1:12pm |
| Parallel Coding | 1:12pm | 1:22pm |
| Debugging/Testing | 1:22pm | 1:27pm |
| | | |
| Debugging/Testing | 2:16pm | 3:37pm |
| | | |
| Debugging/Testing | 1:05pm | 2:53pm |
| Fix Compile Error | 2:53pm | 2:53pm |
| Debugging/Testing | 2:53pm | 3:31pm |
| | | |
| Debugging/Testing | 1:05pm | 1:28pm |
| Performance testing | 1:28pm | 2:09pm |

From ActivityGraph we obtained the programmer's effort distribution, shown in Table 6.

**Table 6. Activity effort distribution and event effort distribution #2**

| Activity | | Event | |
|---|---|---|---|
| Serial coding | 12% | Compile success | 3% |
| Parallel coding | 11% | Compile fail | 0% |
| Performance testing | 7% | viewing | 9% |
| Debugging/testing | 69% | filecommand | 18% |
| Fix compile error | 0% | editing | 28% |
| | | building | 4% |
| | | unknown | 0% |
| | | other | 8% |
| | | running | 30% |

From our results, we obtained the programmer's workflow, shown in Figure 6. The activity categories used by the observer in study #2 were: *thinking, serial coding, parallelizing, debugging, testing, other.* Our activity categories were similar: *serial coding, parallel coding, debugging/testing, performance testing, fix compile error*. The evaluation of our results against the observer results are shown in Table 7. There is quite good agreement between the observer and our analysis. The activity classified as "other" by the observer matches with our performance testing activity.

**Table 7. Activity relationship #2**

| Observer | | SEWA process | | Difference |
|---|---|---|---|---|
| Thinking Serial coding | 15% | Serial coding | 12% | 3% |
| Parallelizing | 8% | Parallel coding | 11% | 3% |
| Other | 7% | Performance testing | 7% | |
| Debugging Testing | 70% | Debugging/testing | 69% | 1% |
| | | Fix compile error | 0% | 0% |

## 5.3 Threats to validity in case studies

While these two small case studies suggest that SEWA can provide accurate estimates of programmer activity times, they contain several threats to validity.

**Internal threats to validity.** The second author had prior familiarity with the validation studies, as he had conducted the original observation studies. While the studies were conducted years earlier and the second author was not directly involved in the data analysis, bias due to familiarity cannot be ruled out.

**External threats to validity.** This process can only be applied on data that can be captured from the programmer's environment. Many relevant activities surely occur during the development process that are not captured by instrumentation or in software repositories (e.g. informal communication among developers).

This study depends very much on the skill of the analyst, and their familiarity with the programming languages, libraries, development tools, and problem being solved. A different analyst may obtain different results. This can be mitigated by using multiple analysts at each stage cross-checking and comparing results.

The programming problems solved in these studies are much, much simpler than the real types of programming problems that are solved on HPC machines, which can take years of development. In our analysis, standardizing the data is done automatically while using the Eight Cs technique to analyze the sequence data is done manually. In this way it is feasible for us to analyze small problems, such as the study #1 and study #2, while it is difficult to analyze large problems, for example it is not realistic to analyze six months of data manually. We could study several days of development for one programmer, but not much more than that.

## 6. Discussion

We believe that our analysis was a reasonably close match to the observer, and therefore has the potential to be useful. In particular, the second study showed for very close agreement with the observer. One source of discrepancy was that the observer had a category called "thinking" that we could not observe from captured data.

It is sometimes difficult to define what should constitute an activity. For example, if the programmer adds some sequential code to a parallel program, it is not clear whether we should define this activity as serial coding or parallel coding. For debugging, we could have chosen to distinguish between serial debugging and parallel debugging, which would make the workflow model more precise but more complex. For researchers who compare how difficult serial vs. parallel debugging is, the separation makes sense, but for our validation study we did not feel it is was necessary. We are more interested in where the programmer spends his time overall in debugging. On the other hand, sometimes we may choose an activity because it is easy to detect, although it may be of little interest in the final analysis. We chose to identify the activity of fixing compile errors because it was relatively easy to detect and we thought it might be useful for later analysis. However, it turned out that such a small fraction of the total time was spent in this activity that it did not add much to our model. Ultimately, there are no "correct" activities. The type of activities that a researcher chooses will be a balance between the research questions of interest, and what is possible to detect with the available data.

During the process it is easy to standardize the data automatically with the event category, but it is difficult to automate the overall process. First, it is hard to implement the heuristics automatically. For example, in study #1, We were able to recognize "buffon" as the program that the developer was writing, but for an automated system this would be harder to detect and hard to comment it as debugging/testing activity.

Another example is how to recognize performance testing. Heuristic #10 describes how to recognize performance testing as repeated execution of program with different nodes and ne source code modification. A human can look at the context to judge if these executions are performance testing or correctness testing, but it is more difficult to automate this kind of judgment. Second, it is difficult to identify programmer activity through automatic analysis of source code changes. A knowledgeable human can make reasonable guesses by looking at source changes in successive compiles: capturing this process is would require the development of an expert system.

From the results we can see that the most time-consuming programmer behavior is debugging/testing. In study #1, 45% of the time is spent on them; in study #2, 69% of the time is spent on them. Building order-of-magnitude estimates on debugging time allows us to estimate the potential impact of improved debugging tools and techniques on overall programmer productivity, and to make return-on-investment arguments for development or acquisition of debugging technologies.

In our study we applied SEWA to reveal where programmers spend their time and reveal their workflow model with respect to serial coding, parallel coding, and testing/debugging. However, the overall SEWA process can be applied by researchers to build workflow models with completely different categories and in different domains. Applying it in new areas and for different activities will require that researchers develop and validate a new set of heuristics.

## 7. Conclusion and future work

Captured software engineering data provides many clues about the kinds of activities programmers engage in. Our work suggests that a researcher can use exploratory-based approach to identify useful heuristics for estimating a programmer's workflow.

The ultimate goal of this work is to automate as much of this process as possible. Except for the standardization step, which already is automated, we still need do two tasks. One is to improve the heuristics and make them objective enough to be distinguished by computers. The other is linking all the analysis steps in the process and making the whole process implemented automatically. This could be done by leveraging existing process mining infrastructures.

We can also build more sophisticated, quantitative models to try perform evaluation and prediction studies. For example, using the data from SEWA we could build timed Markov models such as those built by Funk et al [5].

Finally, we would like to do more studies and look to see how similar the models are across different people solving the same problem. We could apply SEWA in different software domains to know how time is distributed in those domains and how the programmer workflows change across domains.

## 9. Acknowledgments

This research was performed under the DARPA HPCS project under Air Force grant FA8750-05-1-0100 to the University of Maryland.

## 10. References

[1] B. Bederson, J. Grosjean, and J. Meyer. 2004. "Toolkit Design for Interactive Structured Graphics". IEEE Transactions on Software Engineering, 30(8), 2004, pp. 535-546.

[2] J. Cook, and A. Wolf, "Balboa: A framework for event-based process data analysis" in *Proceedings of the International Conference on the Software Process*, 1998.

[3] J. Cook and A. Wolf, "Software process validation: quantitatively measuring the correspondence of a process to a model", *ACM Transactions on Software Engineering Methodology*, 8(2), 1999, pp. 147-176.

[4] K.A. Ericsson and H.A. Simon, *Protocol Analysis: Verbal Reports as Data*, MIT Press, 1993.

[5] A. Funk, J. R. Gilbert, et al. "Modelling Programmer Workflows with Timed Markov Models", *CTWatch Quarterly* 2(4B), Nov 2006.

[6] M. Gardner, "Mathematical Games: The fantastic combinations of John Conway's new solitaire game 'Life'", Scientific American 223, Oct 1970, pp. 120–123.

[7] L. Hochstein, T. Nakamura, et al. "An Environment of Conducting Families of Software Engineering Experiments" In *Advances in Computers* 74, M. Zelkowitz, Ed. Aug. 2008.

[8] L. Hochstein, V.R. Basili, et al. "Combining self-reported and automatic data to improve effort measurement" in *Proceedings of the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE),* 2005.

[9] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd edition, MIT Press, 1999.

[10] P. Johnson and H Kou, "Automated Recognition of Test-Driven Development with Zorro" in *Proceedings of the International Workshop on Software Process*, Shanghai, China, 2006.

[11] P. Johnson, H. Kou, et al. "Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined" in *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, 2003, pp. 641- 646.

[12] F.E. Ritter and J.H. Larkin, "Developing process models as summaries of HCI action sequences", *Human-computer interaction*, 9, 1994, pp-345-383.

[13] A. Ko, R.DeLine, G.Venolia et al, "Information Needs in Collocated Software Development Teams, in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, 2007.

[14] D. Perry, N. Staudenmayer, et al. "Understanding and improving time usage in software development" in *Software Process*, A. Fuggetta and A. Wolf, John Wiley and Sons, 1996.

[15] P.M. Sanderson and C. Fisher, "Exploratory Sequential Data Analysis Continuous Observational Data", *interactions*, 3(2), 1996, pp. 25–34.

[16] E.W. Weisstein, "Buffon-Laplace Needle Problem" from Math World – A Wolfram Web Resource. http://mathworld.wolfram.com/Buffon-LaplaceNeedle Problem.html. Last accessed June 28, 2008.

[17] A. Rozinat and W.M.P. van der Aalst, "Conformance checking of process based on monitoring real behavior", *Information Systems*, 33(1), 2008, pp. 64-95.

[18] M. Scotto, A. Sillitti, G. Succi, and T. Vernazza, "A non-invasive approach to product metrics collection", *Journal of Systems Architecture,* 52(11), 2006, pp. 668-675.

[19] I.D. Coman and A. Sillitti, "An empirical exploratory study on inferring developpers' activities from low-level data", *International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2007, pp. 15-18.