Prototyping A Process Monitoring Experiment

Mark G. Bradac, Dewayne E. Perry and Lawrence G. Votta
AT&T Bell Laboratories

Abstract

Features are often the basic unit of development for a very large software system and represent long-term efforts, spanning up to several years from inception to actual use. Developing an experiment to monitor (by means of sampling) such lengthy processes requires a great deal of care in order to minimize costs and to maximize benefits. Just as prototyping is often a necessary auxiliary step in a large-scale, long-term development effort, so too is prototyping a necessary step in the development of a large-scale, long-term process monitoring experiment. Therefore, we have prototyped our experiment using a representative process and reconstructed data from a large and rich feature development.

This approach has yielded three interesting sets of results. First, we reconstructed a 30 month time diary for the lead engineer of a feature composed of both hardware and software. This data represents the daily state (where the lead engineer spent the majority of his time) for a complete cycle of the development process. Second, we found that we needed to modify our experimental design. Our initial set of states did not represent the data as well as we had hoped. This is exemplified by the fact that the ''other'' category is too large. And finally, the data provides evidence for both a waterfall view and an interactive, cyclic view of software development.

We conclude that the prototyping effort is a necessary part of developing and installing any large-scale process monitoring experiment.

Keywords: prototyping process experiments, process monitoring, process sampling, process analysis

1. Introduction

Features are often the basic unit of development for very large software systems and represent long-term efforts, spanning several years from inception to actual use. Thus, monitoring these lengthy processes is a long-term effort as well. We report here an initial step in the development of an experiment to monitor such processes: that of prototyping the experimental design to monitor a representative process used in developing such features.

In the remainder of Section 1, we provide background information relevant to the general goals of our experiment, present our motivations for both the experiment and our prototype, and discuss related work. In Section 2, we present our experimental design and discuss issues in instrumenting the experiment. In Section 3, we describe our method of prototyping the process monitoring experiment and evaluate the results of the prototype experiment. In Section 4, we discuss the results of our prototype analyses and indicate some interesting aspects of the prototype data. Finally, in Section 5, we present our conclusions.

1.1 Background

Time, like costs, can be viewed as a unit of optimization in improving software development processes. In particular, processes and artifacts that have evolved over time have accreted a variety of inefficiencies. Optimizing the development interval is one way of exposing many of these inefficiencies and reducing costs. While the general goal of most organizations is to maintain or increase the level of quality while reducing costs, there is a need to do this with a well-defined and repeatable process ([10] pp 5, 67ff). The specific goal of our process monitoring experiment is to find ways to reduce the development interval.

Cost, accuracy and concerns about interference are fundamental and interrelated factors in developing experiments to monitor the process of adding features to a large software system. Obviously there is a tradeoff between cost and accuracy in obtaining data about the process. Typically, the higher the accuracy, the higher the cost in obtaining the data. Furthermore, the higher the accuracy, the more likely the process of obtaining that data will be intrusive. This in turn will affect the probability of successfully completing the process — that is, a measure of interference can be viewed as the probability of inhibiting that success.

It is against this background of developmental and experimental factors that we note the utility of modeling development intervals as a queuing network. We believe this is the first time queueing models have been used to describe the development time interval. It is this approach that motivates the need to record the server interval (that is, how the person executing the process is spending his or her time). We suggest there exists a very strong analogy of how software development accomplishes work and how a manufacturing single job shop works.[1]

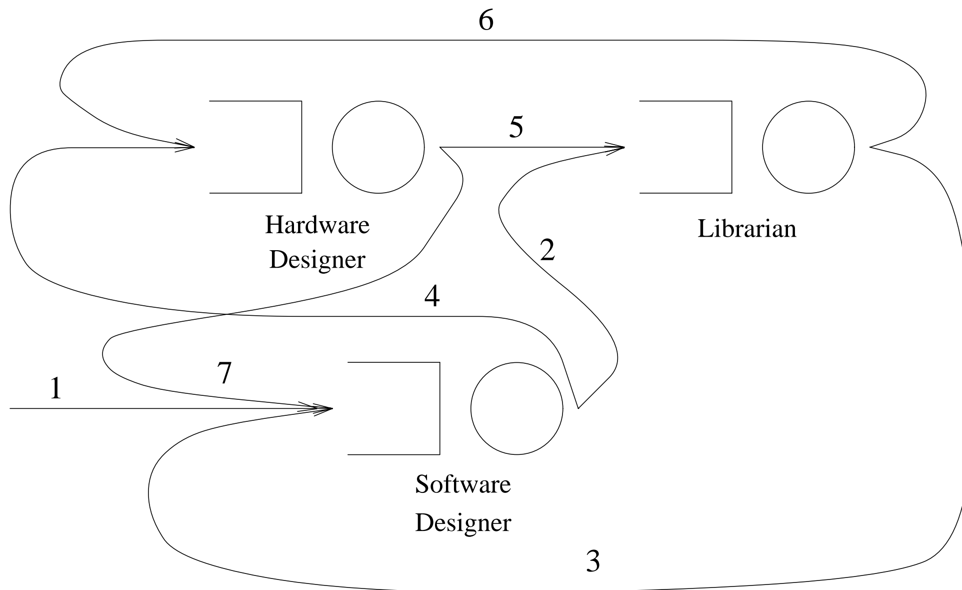

Figure 1A: Software Development Queueing Network Model
This simple queueing network models an example of a software developer's interaction with a project librarian and a hardware developer during the coding of a module.

We illustrate the use of a queueing network as a model for software development by describing the arrival of a new design component to the coding process and the initial work of the software developer. The example starts with the developer idle. A new design component arrives into the queue (figure 1A, arc 1). The developer immediately starts working on the new design component (figure 1B, day 1 and 2 get marked with a ''■''). On the morning of the third day, he discovers the need to consult a hardware interface but finds no reference in his notes. He calls the library (figure 1A, arc 2), finds the library closed because the librarian is at a class, and leaves a message. Since the hardware state information is crucial for algorithm selection, the developer decides to stop working until he gets the hardware description (figure 1B, day 3 gets marked with an ''–'' waiting on the library). He then spends the rest of the day answering mail and catching up on his reading.

The next morning, he finds the hardware design specification in his mail — an efficient librarian answered his request and left the document (figure 1A, arc 3). Using the document, he resolves the algorithm choice and continues work on the remainder of the design (figure 1B, day 4). Late that afternoon, he discovers an inconsistency between an assumption in his notes and the hardware document (which is now two years old). Our developer decides to consult the hardware designer (figure 1A, arc 4) and finds she is no longer familiar with the interface, but will be glad to help. However, she discovers she no longer has the circuit

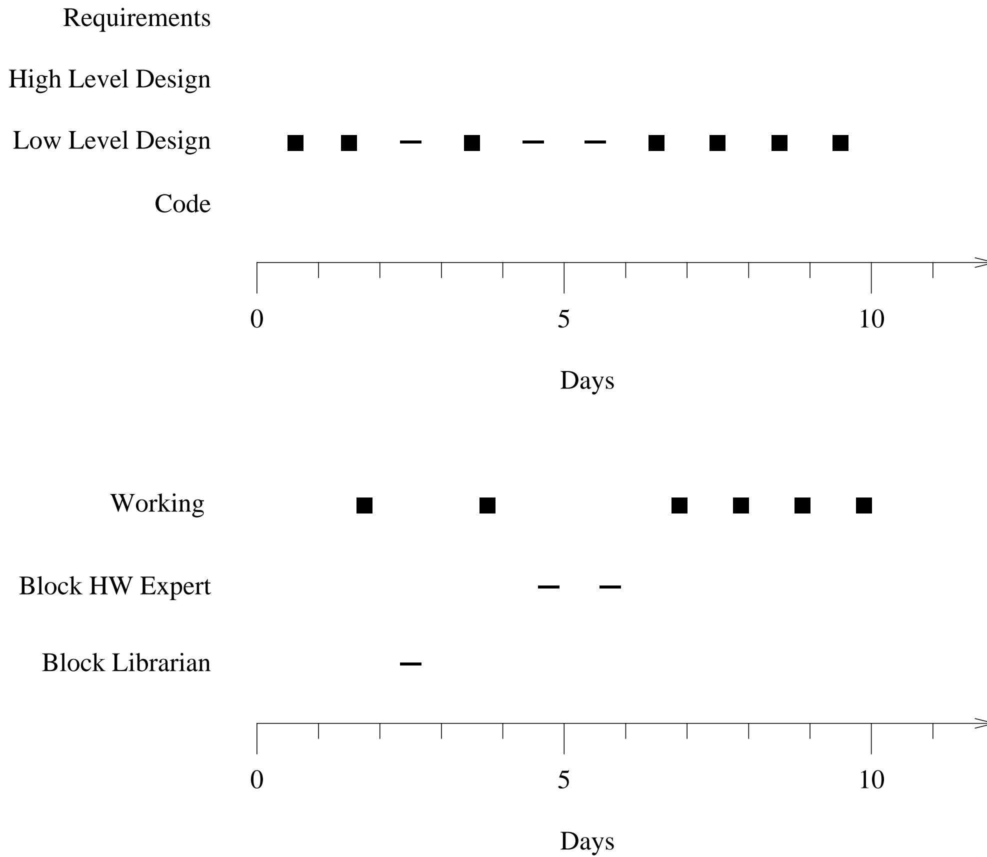

Figure 1B: Plot Of Developer Activity Per Day

These plots show the data collected to characterize what the software developer is doing with the new design component and what state he or she is in The top plot is the software developer's service time distribution for the software development task in units of days and the bottom plot is the service time distribution for the corresponding states.

---

diagrams which the design document describes and must get them from the library (figure 1A, arc 5). While she is waiting on the diagrams, he must find other things to do (figure 1B, day 5 gets marked with an ''–'' waiting on the hardware expert).

Eventually (another day later due to other interruptions), the hardware designer resolves the problem and calls the developer late in the afternoon with the resolution to the conflict (figure 1A, arc 7). He then works for the remaining few hours continuing his implementation (figure 1B, day 6; note that the day is logged as blocked as that was the state for most of the day) and finishes his work in another four days (figure 1B, days 7-10).

1. Therefore, we propose using network queueing models to investigate the software development process.

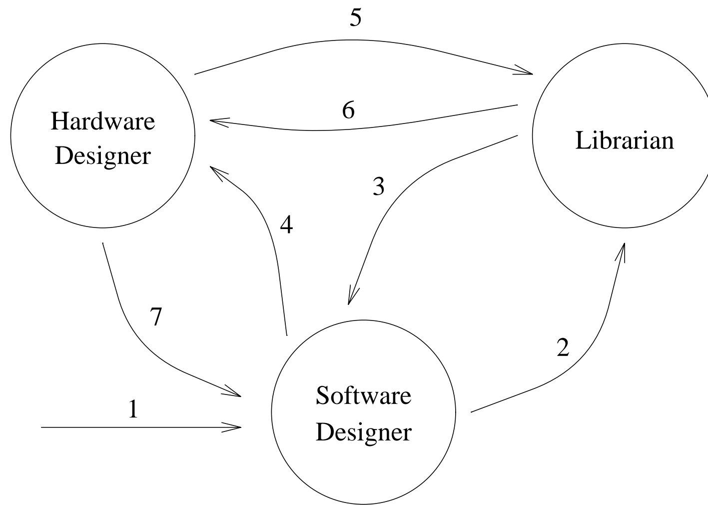

Figure 1C: Communication Network Map

This plot captures the very important details of how the different process executors communicate with each other. Note that one can develop an intuition about these diagrams and how they relate to organizational structure and physical location. The interested reader is directed to chapter 5 of [Allen77]. The arc labeling corresponds to that used in figure 1A. Since we are discussing only a part of a single development, we do not have the frequency information that we expect to gather in subsequent experiments.

Our strategy in this paper will be to assume that queueing networks are a natural fit for modeling process.[2] The sojourn time through the queueing network is just the development interval we would like to understand and control. We know from queueing theory that to calculate the sojourn time in open networks, it is necessary to know the service distributions, the branching probabilities[3], and the queue priority disciplines. However, many approximation techniques exist when information is only partially known, or for instrumentation/cost considerations the information captured is known to be inaccurate [15].

1.2 Motivation

Our motivation for monitoring the current processes is straightforward: we want to find out what people actually do when they add features to a large software system. There are two phrases that have emphasis in this last sentence: ''people'' and ''actually do''. In the first instance, we are interested in what people do and how they interact, not in what tools do and how they interact (though we are interested in how people

---

2. We believe this is a very important topic but, for now, can only offer the observation about the fitness of the modeling technique. As these experiments continue we will investigate a queueing theory framework for modeling the software development process interval.

3. This is the network map in figure 1C with the additional information of how probable each arc from a given node would be taken.

interact with tools). What people do and how they interact are of paramount importance in engineering processes [1].

In the second instance, we are concerned about what they actually do as opposed to what they are intended to do or what they are thought to do. We want to understand how people progress through their activities (*a la* Guindon's study of software design [8]) and where and how they are hindered from making that progress in those activities.

With features as the unit of development, it is important to understand what people do when developing a feature, how they interact within a single feature development, and how the different development groups interact with each other in developing several features concurrently. In particular, do these individual and group interactions depend on the kind, size and complexity of the features? Or are they independent of these factors?

The purpose of this experiment is to provide an understanding of the feature development processes. We then will use this understanding as the basis both for accurate descriptions of, and for substantive improvements to, these processes.

Our motivation for prototyping the experimental design is also straightforward: the experiment will be a long-term effort and we want to work out as many wrinkles as possible before committing to the actual experiment. The situation here differs from that of Basili and Weiss [3][4] in the following way: they already had a (costly) mechanism in place within which they proposed to do their experimentation; we have no such mechanism (costly or otherwise) in place. In fact, the desired result of the experiment is to install just such a mechanism.

Our approach, then, is to prototype the experiment by means of a case study. By so doing, we demonstrate how to use a prototyped experiment to tune the instrumentation and to hone the experimental purpose before we do the actual experiment. This enables us to short-circuit some of the usual experiment pitfalls and to increase the efficacy of the experiment.

Analogous to our need to understand the interactions among people and groups, we need to understand the interactions among various aspects of the experiment. Our general problem in the experiment is that we have a large number of variables. Are they all essential, and if not, which ones are? Are we missing something important in the experiment? For example, if the various aspects of the hardware processes dominate the entire process, then reducing the wait time in the software process will not have the intended impact on the interval. It is just these kinds of factors and interactions we want to understand before

4. Basili and Weiss emphasize validation of data collection. We completely agree with that position. We would further maintain that this position is valid for the entire process, not just the part that they study (namely, from coding on). We feel prototyping an experiment is a valuable extension to their work when, for whatever reason, neither the methodology nor instrumentation is in place.

conducting the full experiment.

There are two further concerns with respect to prototyping the experiment that are important. First we want to minimize the impact on development — that is, we want to minimize the cost of the experiment, minimize the interference that may result from the experiment, and maximize the accuracy of the data collected. Second, we want to illustrate the experiment and its possible results. We then gain early feedback from those participating in the experiment. We also motivate the participants by showing various kinds of anticipated results.

1.3 Related Work

One of the points of discussion at the 5th International Software Process Workshop was the need to look at other fields to see how they solved the problems of modeling their processes [17]. We note that queueing networks have been extensively used for modeling complex manufacturing processes [15].

Virtually all the work reported at the various process workshops (see [20], [27], [6], [26], [17], [12], and [25]) and the 1st and 2nd International Conferences on the Software Process [7, 16] is concerned with process formalisms, analysis and support. There are several strands of related work that we consider important in reducing task intervals: Kellner's use of Statemate [9] to provide a management perspective on the modeled processes [13]; Perry's policy- and product-induced partial ordering of process activities [18]; Kaiser and Barghouti's implementation of concurrent rule-chains in automating tool interactions [2]; and Mi and Scacchi's inclusion of roles and resources as basic entities in the Articulator Model [Wi91].

A common thread in these discussions is that of providing support for monitoring and measuring processes. There are several frameworks for measurement-guided software development. The TAME project [4] and the Amadeus system [23] are two such frameworks (the latter aimed at providing measurement and feedback within a process-centered environment [24]). In general, their emphasis is on automating measurement and control of evolving products.[5] This approach assumes that we know what needs to be measured to control the process at the desired level of precision within the desired cost constraints. We believe that we have not reached that level of maturity in the measurement and control of software processes.

The Software Engineering Laboratory [5] has focused on experiments of various kinds to determine the effects of different languages, methods and tools on the software development process. They have not, to our knowledge, taken our approach to monitoring processes or to prototyping their experiments.

5. More specifically related to providing information about processes themselves have been the discussions of the DARPA Prototech Process Virtual Machine (PVM) Working Group [21]. The work there, however, is still in an the exploratory stage, but it is envisioned that the resulting framework will provide basic facilities for monitoring various aspects of the processes executed within the PVM framework.

The only work of which we know that is similar to ours is that of Wolf and Rosenblum [28]. In their approach, they monitor the processes *in situ*. While we are similar in our emphases on various events and intervals, we differ in several important respects, both in experimental approach and in the subject of our experiments. First, our means of capturing the process data are quite different. Second, the process they are monitoring (the load build process) is a very small one that is repeated both regularly and frequently. Finally, they are monitoring a process that deals with an entire system while we are dealing with individual feature developments within the context of an entire system.

2. Experimental Design and Instrumentation

We first describe the design of the experiment, then discuss several important aspects of instrumentation (namely, cost, precision, and accuracy), and finally, discuss the problems of repeatability and reproducibility.

2.1 Design

There are two important threats to the external validity [11] of these experiments: (1) the relevance of the experiment to the real world and (2) the generalizability of the results to all subjects. Both of these threats are addressed by using a real software development. Furthermore, the process model used in the experiment is the process model the developers use.

The approach we use in this experiment is to sample process activities on a periodic basis. We use the samples to build a database of how time was spent throughout the development of various selected features. We claim the advantages of this approach are its simplicity and its low cost.

A process is characterized by a set of tasks and a set of states. The tasks define the various activities within the process that are of interest and that will be sampled in the experiment. The states represent either progress within a task or lack of progress (that is, where the task is blocked for some reason).

The participant is reminded on a periodic basis (daily is the usual periodicity) to remotely log the activity of the previous day. The tool presents the user with a menu of tasks. Once the task has been selected, the tool presents the user with the menu of states for that task. The user, then, selects the most appropriate one. The intent is to sample the most important aspect of the task on the previous day. The tool automatically provides an ‘‘other’’ category for both tasks and states. If ‘‘other ’’ is selected, the tool then elicits a textual description to clarify what was done or how the person was blocked. This latter facility enables us to tune the experiment while in process if the ‘‘other’’ category becomes too large.

These basic mechanisms in the experiment are:

- the email system to remind users to log the previous days tasks and the state of progress within that task;
- a menu-based tool activated by remote login throughout the network of development machines to capture by means of reminder prompts the previous days work; and
- an online database to accumulate the data entered by the various user that may be queried at any time by the process experimenters.

The experiment will range over a set of feature developments chosen to balance their characteristics with respect to their kind, size and complexity. Since there will be on the order of three to four different process monitored over these developments, we should be able to gather data that is sufficient to be representative of the kinds of different software that is developed on this project.

2.2 Cost, Precision and Accuracy

The experiment incurs only very minor overhead. First, we already have email and general logins on all machines as a part of the development environment. Second, the sampling tool already exists as a part of the change management environment and requires only minor modifications to make it applicable to the approach needed for the experiment. Finally, training for use of the modified tool is small as the tool is both known understood, and self-documenting. Training is instead focused on clarifying the process tasks and states.

We consider precision to be the sampling granularity. Just what that granularity should be is a trade-off between cost and accuracy. While hourly data would give us a more detailed understanding of a process, we consider it to be too fine-grained, particularly as most of the feature development processes last longer than 30 days. Moreover, the more often data is sampled, the more intrusive the sampling becomes. Thus, a daily sampling rate seems to be a useful initial compromise.

We have several problems with regard to the accuracy of the data we are gathering. First, there are the typical problems with sampling compared to gathering complete data. However, we expect these problems to be lessened by the large set of samples that we will gather. Second, people often do multiple tasks concurrently and do multiple things during the course of a workday. However, this is ameliorated by the fact that we monitor per feature development (which will differentiate the effort of a developer working on several features concurrently) and that we have a blocking category for other assignments (which will differentiate a developer doing things other than working on this feature). Furthermore, we assume that developers really only do one or two things per day per process — that is, that their time is not overly fragmented.

2.3 Repeatability and Reproducibility

Ideally, a developer should give the same response to the tool when identical or nearly similar progress (or lack of it) has been made. We believe we will achieve a sufficient level of repeatability in the experiment for the following reasons. First, the language used in the categories and explanations is that of the developers and is well-understood (and in fact designed by the process engineers who are also developers). Second, there will be close monitoring of, interaction with, and training of, the users in the early stages of the experiment. Finally, we restrict the number of categories to seven plus or minus two to minimize the possible discrimination variance [14].

Equally ideally, different developers within the same development should provide the same sample path. Obviously this will not be true, as this is a stochastic process and there will be variances among sample paths through the queueing network. For example, blocking will vary depending on whether an expert is available, on the phone, or on vacation. However, for some expected value sense, we should see the same behavior. Another factor contributing to variance in the experiment is the low level of maturity of the processes involved. This variance comes from the continuous ‘‘tinkering’’ with the process, either explicitly by management or implicitly by the developers as they improvise less well-defined or understood parts of the process.

3. The Prototype Experiment

We need three things to prototype the experiment: a process, a development, and a set of analyses. For the process, we selected a well-understood one that is a standard development process. The advantages are straightforward: we, as experimenters, understand the process as well as developers and are unlikely to misinterpret the experimental results. For the development, we selected a relatively large feature development for two reasons: first, we thought that we would be able to reconstruct fairly accurate data; and second, we felt that a large complex feature would stress the experiment in such a way that if there were inherent problems, we would see them. For the analyses, we selected a set of basic views of the data to consider both the blocked and working states of the process.

We first discuss the process, development and analyses of our prototype. We next discuss the results of our analyses and their effect on the structure of the experiment. Finally we consider the costs of the prototype experiment.

3.1 The Development, Process, and Analyses

The development for this experiment implements a feature comprised of both hardware and software. The component studied is the diagnostic software that initializes the hardware, isolates faults, and interfaces with the rest of the system.

Task	*State*
Unassigned	Working the Process
Estimate and Investigate	Documentation
Plan Development	Reworking the Process
Requirements	Reworking the Documentation
High Level Design	
Low Level Design	Waiting on the Lab
Write Test Plans	Waiting on an Expert
Code	Waiting on a Review
Inspections and Walk-throughs	Waiting on Hardware
Low Level Test	Waiting on Software
High Level Test	Waiting on Documentation
Costumer Documentation	Waiting Other
Support	
Postmortem	
Weekend	

Table 1: Tasks and States

We initially characterized the process in terms of fifteen tasks and eleven states. The states represent a binary choice: either making progress or waiting. Within those two choices, there are four and seven states respectively. Table 1 lists the initial tasks and states (with a blank line delineating the two sets of state choices).

We used log books, personal diaries, and project management notebooks to reconstruct a set of data to represent one developer's experience with the selected process. The benefits of doing this are obvious: we have data that reflects actual experience of the project. The primary disadvantage is the temporal averaging that results in the loss of clarity and acuity. This disadvantage is the reason for not using reconstruction as the primary means for gaining insight into the process. We note that reconstruction does, however, provide a useful means of forming hypotheses about the relevant processes.

Having reconstructed a set of data, we then decided on a basic set of analyses to provide us insight into the development process: the distributions of feature development tasks over time, the distribution of states over time, the level of blocking for each task, the breakdown of work and rework for each task, a point plot of tasks over time, and a point plot of states over time. The first two analyses represent results we use to evaluate our prototype while the remaining analyses represent results we use to form conjectures about the eventual results of the experiment.

3.2 Instrument Evaluation

Figure 2 shows the percentages of time spent in each task over the life of the development. Weekend data

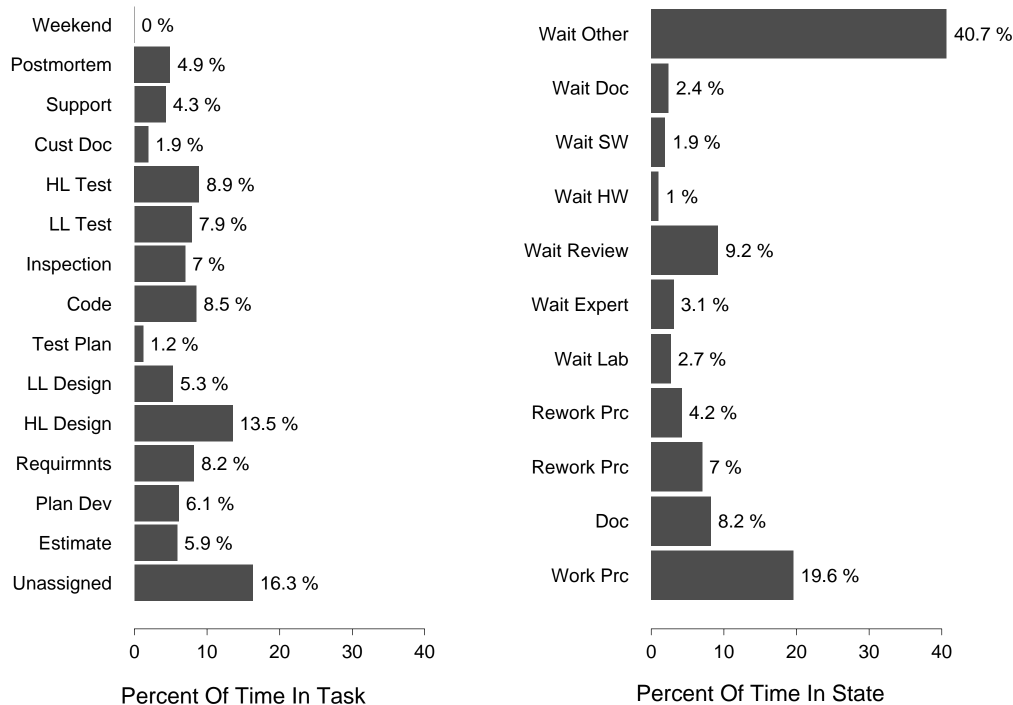

Figure 2: Feature Development Percent Time in Task and State

These barplots show the percent of time spent in each task and state. All percentages are calculated with weekend days removed.

---

was originally included because there might be times when people worked on weekends. In this case, however, the weekend data contained nothing useful and only served to skew the relationships between the other tasks. Hence, figure 2 represents the data excluding weekends. Note that the unassigned category dominates the time spent.[6]

The percentages of time spent in each state over the life of the development are also shown in figure 2. Again weekend data skewed the relationships and has been removed. Note that the ‘‘wait other’’ state is overwhelmingly dominant. This points out a weakness in our characterization of the process and needs to be corrected. Our initial thought was to collapse several of the lightly populated states into one encompassing state — for example, to fold waiting on the laboratory, hardware and software into waiting on resources and to fold waiting on experts and documentation into waiting on information. However, discussions with development management convinced us that these categories needed to remain distinct

6. People are specifically unassigned from a particular feature development to make them available to respond to unanticipated requests from customers of the system.

Task	*State*
Unassigned	Working the Process
Estimate and Investigate	Documentation
Plan Development	Reworking the Process
Requirements	Reworking the Documentation
High Level Design	
Low Level Design	Waiting on the Lab
Write Test Plans	Waiting on an Expert
Code	Waiting on a Review
Inspections and Walk-throughs	Waiting on Hardware
Low Level Test	Waiting on Software
High Level Test	Waiting on Documentation
Costumer Documentation	
Support	Not Working: Training
Postmortem	Not Working: Other Assignment
Weekend	Not Working: Vacation and Holiday
	Not Working: Weekend
	Other

Table 2: Tasks and Amended States

since lessening their individual effects requires interactions with different organizations. Instead, we have corrected this part of the experimental design by dividing the states into a trinary choice rather than a binary one: not working, making progress, and blocked waiting to make progress. Several large blocks of time in the ''wait oth'' category were spent on training, vacation and holidays, and working on other assignments. Thus we have refined our design by extending the states to reduce the amount of unspecified other ways of spending time in the process.

Table 2 illustrates the way in which we have extended states from a binary choice to a trinary one. The additional states represent the third choice of not working the process by either working another development, attending training, going on vacation, or representing the lack of effort due to the weekend.

We note that the ''wait other'' state is intended to serve primarily as a safety net in both the prototype experiment and subsequent experiments. While our emphasis here has been primarily on progress and lack of progress,[7] our state categorizations are of necessity incomplete, since we wish to keep the number of them within an easily managed bound. Subsequent experimentation may necessitate further refinement of

7. We also note that our emphasis on progress versus lack of progress is not the only categorization of states that is possible. For example, one might want to probe deeper into what kinds of progress are being made, or what forms of discovery are necessary to making progress, etc.

this category.

As the process we used was both well understood and well-established, we had no problems in deriving an appropriate set of tasks. However, for processes that are not so well-defined, this aspect may be an important part of a prototyping effort.

3.3 Cost of the Prototype Experiment

The prototype effort took about four person-months of effort. Approximately 2 weeks were spent reconstructing the development data. The remainder of the time was spent analyzing the data, discussing the analyses with both developers and development management, and reviewing the task and state structures with representative developers.

We consider this level of effort to be very cost effective for an experiment that will involve a variety of feature developments over a period of several years.

4. Analysis and Resulting Hypotheses

We present the analyses of our prototype with the following caveats: first, the reconstructed development data represents only one sample path through the process — that is, it is one instance of the process; second, the accuracy of the data is open to question because of its retrospective nature — some information undoubtedly has been lost in the lapse of time and some of the task and state categorizations are judgments that may not be accurate because of this loss of information. None the less, we present them because they are consistent with our experience as software developers and very suggestive about software development processes in general.

Figure 3 shows the relationship between time spent productively and time spent waiting. In almost every case the time spend waiting exceeds that of productive work. We also note that 60% of the time was spent waiting rather than being productive. While there may well be other factors to consider, it seems clear that one important way of improving the process is to reduce significantly the number of days in blocking states. The utility of this conjecture depends on the degree of multiplexing within these processes. Clearly if the global level of blocking is consonant with this local level, the conjecture will hold.

Of the time spent productively, approximately 27% of the total process time was spent working and reworking the process, and approximately 13% of the time was spent writing and rewriting documentation. Figure 4 delineates the relative amounts of time spent making progress. About one third as much time was spent reworking the process as working it; about one half as much time was spent rewriting documentation as was spent initially rewriting it. Can the rework and rewriting be reduced or is this the basic cost of

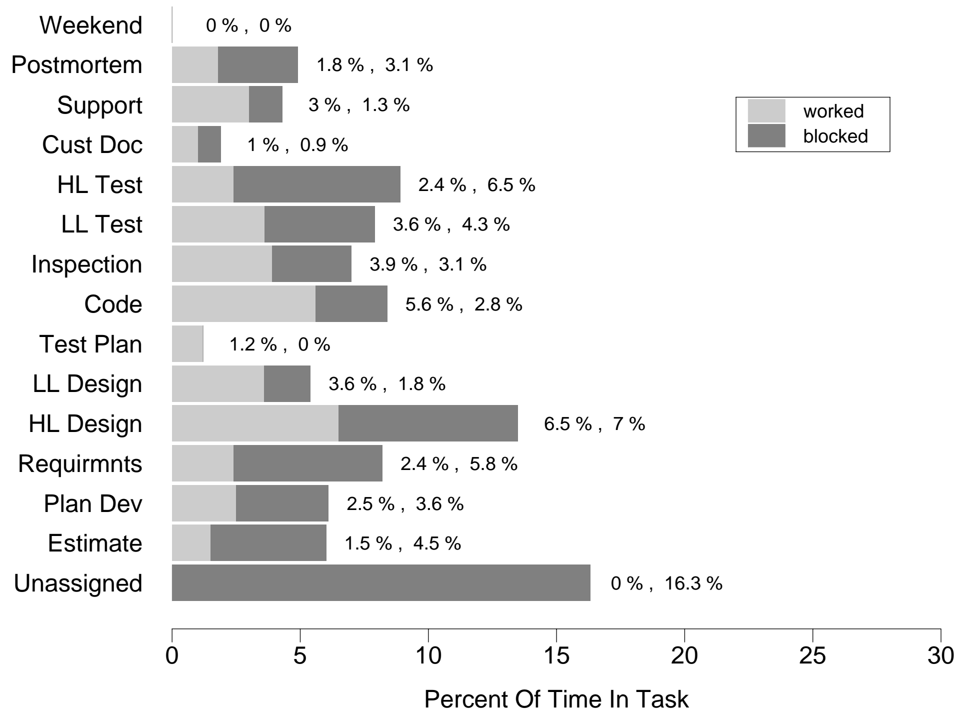

Figure 3: Feature Development Percent Time In Worked And Blocked States
This histogram displays the percent of time worked and percent of time blocked horizontally stacked to its right for each task in the process. The sum of the two percents is the total percentage of time spent in that task of the process. We observe that for almost every case the process is blocked more often than worked.

---

evolutionary software development? If they can be reduced, what are the factors that contribute to rework and rewriting that can be eliminated?

The other thing to note about the integration of figures 2 and 3 is that blocking tends to be more prevalent at the beginning and at the end of the process. Figure 5 illustrates the blocking relationships between the various tasks in the process, indicating the relative weight of each task (that is, how much time was spent in that tsak) as well as the weight of the blocking factors (that is, how much time was spent waiting). The middle of the process (low level design, test planning, and coding) tend not to be interrupted by waiting on other factors. On the one hand, this is not particularly surprising. Clearly, one should attack the blocking factors in the requirements, high level design, and high level test phases of the process since they are more heavily weighted. It will be interesting to see if this conjecture represented in this graph holds over a wide variety of developments.

Figures 6A and 6B represent an early part of the development and indicate the task and state, respectively, for each day. It is worth noting that the first part of the development is almost a pure waterfall process,

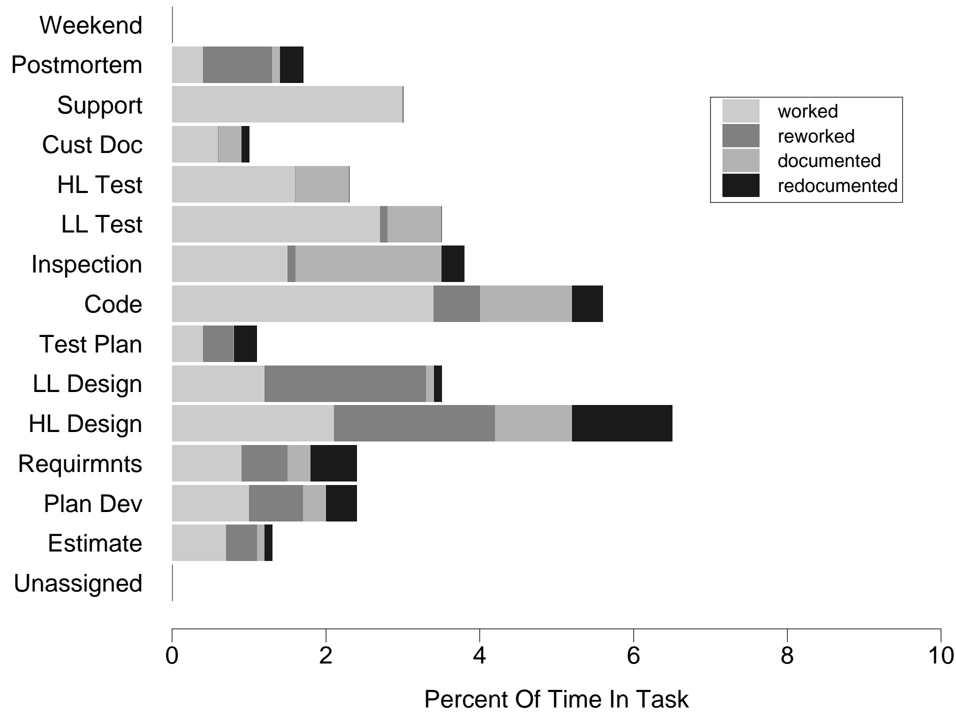

Figure 4: Feature Development Percent Time In Worked States

This histogram displays the percent of time worked, reworked, documented, and redocumented horizontally stacked to the right for each state in the process. The sum of the four percents is the total percentage of time spent in the worked states of each task. For example, from Figure 3 the high level design (HL Des) task is in the worked states 6.5% of the entire process exactly the length of the bar in this figure.

---

moving first through the plan development task and then to the requirements. It will certainly be an interesting fact if this holds over a large class of processes and developments. The important question then will be the source of this linearization.

The second thing worth noting are the lengthy blocking times in this phase of the process. In particular, note the time spent waiting on reviews — four to seven days in a row. Not surprisingly, waiting for reviews dominated both the beginning and the end of the process, but was relatively infrequent in the middle. The other important factor here is the ‘‘wait oth’’ category. It dominates even the review category.

Figures 7A and 7B representing a later part of the process, however, show a completely different overall process: various tasks are intermixed, alternating between four and five different tasks and the various blocking factors are intermixed as well, with none of them taking more than three days and most on the order of one or two days. This reflects more the kind of process that we would expect and the kind that Guindon has shown to be prevalent in her work on the design process [8].

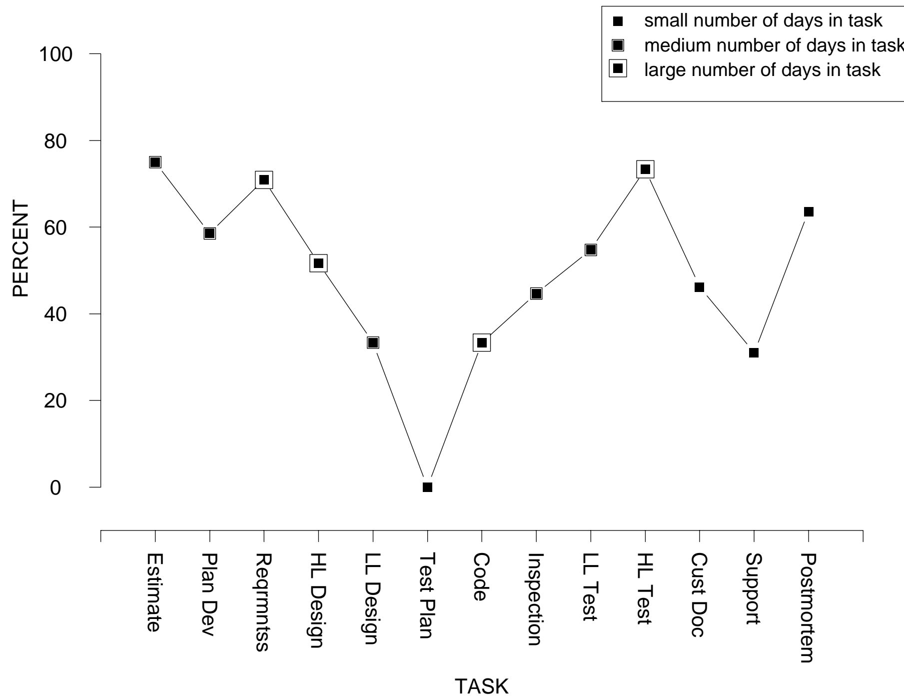

Figure 5: Percentage Of Time Blocked Per Task

The plot shows the percent of time blocked per task. We have encoded the size of each task as small, medium, or a large number of days. Therefore, we note that the high percentage blocked on *Prj Rtrs* is not as important as that of *Reqs*.

We reiterate our caveat about this data: while it is real data, it is reconstructed data of only one instance of the process with some blurring of the accuracy because of retrospection. We feel, however, that there are some intriguing conjectures about our feature development processes that we hope to validate with subsequent experiments.

5. Conclusion

We have taken a simple approach to gaining information about existing feature development processes: sampling on a daily basis the activities performed by the people executing the process. While there are some disadvantages with this sampling approach, it is cost effective and unintrusive, and yields fairly accurate and precise information about how the process is actually performed by the people involved.[8] We

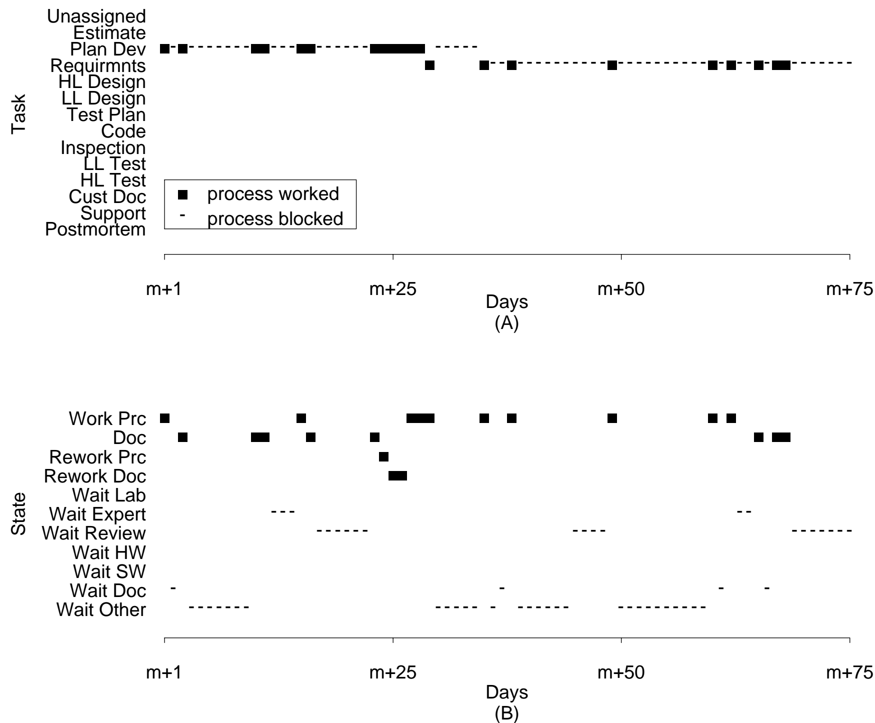

Figure 6: Task And State Developer Data Per Day

The two plots are the data collected from the developer every day. In plot A, we display the task versus day for the developer executing the process. The square character indicates the developer was in an unblocked state, the minus character indicates the developer was in a blocked state. In plot B, we display the state versus day with the worked and blocked states encoded the same as plot A. We note the orderly, *waterfall like* progress of the project and the relatively large consecutive number of days in blocked states.

---

have also provided a formal context for the experiment by considering software development to be an example of a queueing network. To illustrate the appropriateness of this model, we provided a familiar example of a part of a development process.

Given that this sequence of experiments is a long-term effort and that we do not have an existing mechanism in place for monitoring software processes, we claim that it is necessary to prototype our experiment. The prototype is a precondition to a major investment of effort on the part of feature development teams and of resources that should not be squandered. Last, but not least, the prototype is a

---

8. For further information about our experiments, see ‘‘People, Organizations, and Process Improvement’’ [19].

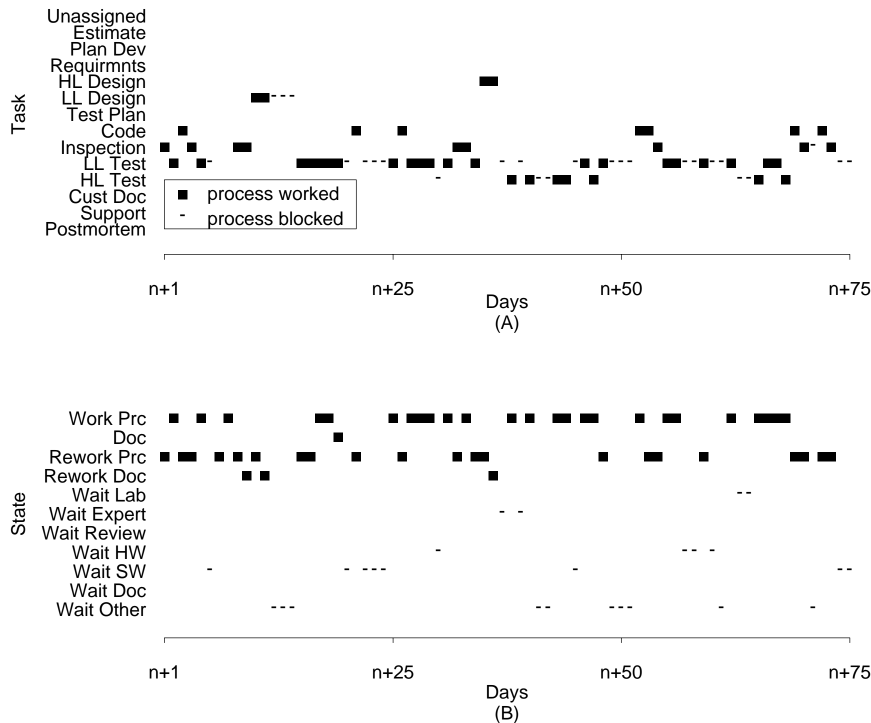

Figure 7: Task And State Developer Data Per Day

The two plots are the data collected from the same developer every day but later in the project than figure 6. We note the significant difference in behavior of the data! The developer is no longer executing the process in an orderly, *waterfall like* way. The average number of consecutive blocked days is much smaller than in figure 6.

---

means of motivation (as an illustration of what can be accomplished) for the company to spend the money and the people to spend the time on process monitoring experiments.

As a part of the prototype experiment, we illustrated some interesting ways to analyze the data, and refined our experimental categories for this particular process (for example, to ignore the weekend state, and refine the ‘‘other’’ blocking state).

We conclude that just as prototyping is often a necessary auxiliary step in a large-scale, long-term development effort, so to is prototyping a necessary step in the development of a large-scale, long-term process monitoring experiment.

Our expectations for further experimentation are that we will obtain extremely useful data about our feature development processes and that some of our conjectures based on our reconstructed data will be validated. These results will be significantly more conclusive than our prototyped results because these experiments will range over both multiple instances of the monitored processes and multiple features carefully mixed as

to size, kind and complexity.

Acknowledgements

We thank Al Barshefsky, Dave Fredericks, and Wen Lin for there unflagging effort in coordinating the experiment within the development process teams, Todd Livesey and Rich Feich for the support of (and willingness to adapt) the sampling tool, Muerl Smith for his insight into the blocking factors and their relationship to organizational politics, and Ward Whitt for his help with our queueing network model.

References

[1] Thomas J. Allen, *Managing the Flow of Technology*. Cambridge Mass: MIT Press, 1977.

[2] Naser S. Barghouti and Gail E. Kaiser, ''Modeling Concurrency in Rule-Based Development Environments'', *IEEE-Expert*, December 1990. pp 15-27.

[3] Victor R. Basili and David M. Weiss, ''A Methodology for Collecting Valid Software Engineering Data'', *IEEE Transactions on Software Engineering*, SE-10:6 (November 1984). pp 728-738.

[4] Victor R. Basili and H. Dieter Rombach. ''The TAME project: Towards improvement-oriented software environments'', *IEEE Transactions on Software Engineering*, SE-14:6 (June 1988). pp 758-773.

[5] Victor Basili, Gianluigi Caldiera, Frank McGarry, Rose Pajerski, Gerald Page and Sharon Waligora. ''The Software Engineering Laboratory — an Operational Software Experience Factory'', *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, May 1992. pp 370-381.

[6] Mark Dowson, editor. *Iteration in the Software Process: Proceedings of the 3rd -International Software Process Workshop,* Breckenridge, Colorado, USA, November 1986. IEEE Computer Society Press, 1987.

[7] Mark Dowson, editor. *Proceedings of the 1st International Conference on the Software Process: Manufacturing Complex Systems,* Redondo Beach, CA, USA, October 1991, IEEE Computer Society Press, 1991.

[8] Raymonde Guindon, ''Designing the Design Process: Exploiting Opportunistic Thoughts'', *Human—Computer Interaction*, vol 5 (1990). pp 305-344.

[9] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. ''STATEMATE: a working Environment for the Development of Complex Reactive Systems'', *Proceedings of the 10th International Conference on Software Engineering* Singapore, April 1988. pp 396-406.

[10] Watts S. Humphrey, *Managing the Software Process*. Reading Mass: Addison-Wesley, 1989.

[11] Charles M. Judd, Eliot R. Smith, Louise H. Kidder. *Research Methods in Social Relations*. 6th Edition. Harcourt Brace Jovanovich, 1991.

[12] Takuya Katayama, editor. *Support for the Software Process: Proceedings of the 6th International Software Process Workshop*, Hakodate, Japan, October 1990. IEEE Computer Society Press.

[13] Marc I. Kellner, ''Software Process Modeling Support for Management Planning and Control''. in [7]. pp 8-28.

[14] George A. Miller, ''The Magic Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information'', *Psychology Review*, 63:2 (March 1956). pp 81-97.

[15] [15], Ardavan Nozari and Ward Whitt, "Estimating Average Production Intervals Using Inventory Measurements: Little's Law For Partially Observable Processes," *Operations Research*, 36:2 (March-April 1988). p 308.

[16] Leon J. Osterweil, editor. *Proceedings of the 1st International Conference on the Software Process: Continuous Software Process Improvement,* Berlin, Germany, February 1993. IEEE Computer Society Press, 1993.

[17] Dewayne E. Perry. *Experience with Software Process Models: Proceedings of the 5th International Software Process Workshop* Kennebunkport ME, USA, October 1989. IEEE Computer Society Press, 1990.

[18] Dewayne E. Perry. ''Policy-Directed Coordination and Cooperation'', in [25].

[19] Dewayne E. Perry, Nancy A. Staudenmayer, and Lawrence G. Votta, ''People, Organizations, and Process Improvement'', *IEEE Software*, 11:4 (July 1994), pp 36-45.

[20] Colin Potts, editor. *Proceedings of the Software Process Workshop*, Egham, Surrey, UK, February 1984. IEEE Computer Society Press, 1984.

[21] Discussions of the Process Virtual Machine Working Group, DARPA Prototech Community, Various working group meetings in Boston, Los Angeles and Sante Fe during 1991 and 1992.

[22] Peiwei Mi and Walt Scacchi. ''Modeling Articulation Work in Software Engineering Processes'', in [7]. pp 188-201.kM

[23] Richard W. Selby. Adam A. Porter. Doug C. Schmidt, and Jim Berney. ''Metric-Driven Analysis and Feedback Systems for Enabling Empirically Guided Software Development'', *Proceedings of the 13th International Conference on Software Engineering*, Austin TX, May 1991. pp 288-298.

[24] R. N. Taylor, R. W. Selby, M. Young, F. C. Belz, L. A. Clarke, J. C. Wileden, L. Osterweil, A. L. Wolf. ''Foundations for the Arcadia Environment Architecture''. *ACM SIGSOFT '88: Third Symposium on Software Development Environments*, 28-30 November 1988, Boston MA. pp 1-13. *SIGSOFT Software Engineering Notes*, 13:5 (November 1988). *SIGPLAN Notices*, 24:2 (February 1989).

[25] Ian Thomas, editor. *Proceedings of the 7th International Software Process Workshop*, Yountville CA, USA, October 1991. IEEE Computer Society Press.

[26] Colin Tully, editor *Representing and Enacting the Software Process: Proceedings of the 4th International Software Process Workshop,* Moretonhampstead, Devon, UK, May 1988. *ACM SIGSOFT Software Engineering Notes,* Volume 14, Number 4 (June 1989).

[27] Jack C. Wileden and Mark Dowson, editors. *Software Process and Software Environments: Proceedings of the 2nd International Software Process Workshop,* Coto de Caza, California, USA, March 1985. *ACM SIGSOFT Software Engineering Notes,* Volume 11, Number 4 (August 1986).

[28] Alexander L. Wolf and David S. Rosenblum ‘‘A Study in Software Process Capture and Analysis’’. *2nd International Conference on the Software Process*, Berlin, Germany, February 1993.