

Toward a Task Model of Concurrent Software Maintenance*

Scott D. Fleming[†]
sdf@cse.msu.edu

R. E. K. Stirewalt[†]
stire@cse.msu.edu

Eileen Kraemer^{†‡}
eileen@cs.uga.edu

[†]Dept. of Computer Science and Engineering
Michigan State University
East Lansing, Michigan, USA 48824

[‡]Department of Computer Science
University of Georgia
Athens, Georgia, USA 30602-7404

ABSTRACT

This paper describes a first step toward developing a methodology for the maintenance of concurrent software that incorporates best practices in design and verification. Specifically, we describe our plan for using the think-aloud method to study the strategies, goals, and intentions of contemporary practitioners engaged in the maintenance of concurrent software. The method will yield a task model that details the specific tasks practitioners undertake while so engaged. Initially, we will conduct the study with graduate students in a formal-methods course at Michigan State University.

1. INTRODUCTION

Developing and maintaining concurrent software systems is notoriously difficult. Numerous approaches and best practices have been developed to support design and verification activities in this context (e.g., [1, 7, 6]). However, because maintenance costs tend to dominate design and development costs [5], practitioners may hesitate to invest the effort required to properly apply these approaches—especially those that use specialized modeling notations. Long term, we seek to develop a methodology for the maintenance of concurrent software that incorporates best practices in design and verification. As a first step, we seek to better understand the strategies, goals, and intentions of practitioners engaged in the maintenance of concurrent software. Accordingly, we propose to develop a *task model* that details the specific tasks practitioners currently undertake while so engaged.

To construct this model, we will study programmers in the act of extending an existing concurrent software system, using the *think-aloud method*, which is a rigorous empirical technique used to obtain a model of the cognitive processes that take place during an activity or to test the validity of a proposed model [4, 12]. We intend to apply the method with one set of participants in order to develop the model and then apply it with an independent set in order to evaluate the model. This approach is consistent with Newell and Simon [8].

Other researchers have studied programmers as they perform maintenance activities. Soloway and Letovsky measured the bene-

*Funded in part by NSF grant CCF 0702667, the Office of Naval Research under Grant No. N00014-01-1-0744, and LogicBlox Inc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WEASELTech'07, November 6, 2007, Atlanta, Georgia, USA.
Copyright 2007 ACM 978-1-59593-880-0/07/0011 ...\$5.00.

fits of documenting the existence of delocalized plans [11]. More recently, Sillito et al. developed a taxonomy of questions that programmers ask while performing general maintenance activities [10]. Previously published models of software maintenance tasks (e.g., [11, 2]) were developed in the context of sequential software. In the context of general software development, Newman et al. identified questions programmers ask about a program's concurrent implementation [9].

These existing maintenance task models and the question taxonomies should lend insight into the formation of our model. However, they do not address problems that are specific to concurrent software maintenance. For example, how one performs testing in the context of nondeterministic thread scheduling or how one assesses the impact of changes with respect to implicit synchronization policies and locking disciplines. Such problems are addressed in the context of design and verification activities using the methods mentioned earlier (i.e., [1, 7, 6]). We believe the processes prescribed by these methods could be used to identify and categorize subtasks related to concurrency concerns.

We see the prior work in this area as containing a wealth of models, each of which addresses part of the problem of concurrent software maintenance, but none of which address the whole problem. Our approach is to evaluate each of these models on a unified set of data, to gauge each of the models for fitness, and to then formulate an integrated task model based on these results. Presumably, none of the models will be complete and “holes” created by one might be filled or partially filled by others. Ideally, the holes left by a general software maintenance model will be nicely filled by the subtasks from concurrent design and verification in which case model integration involves merely refining an existing task model with subtasks specific to concurrency. It may also be the case that the existing models are not so well encapsulated and that a new task model will emerge. In any event, a detailed analysis of maintenance activities in this context will permit us to assess whether existing task models can be further articulated or whether the added complexity of concurrency necessitates a new task model.

2. BACKGROUND

Following the think-aloud method, investigators observe participants engaging in some activity and collect *think-aloud protocols* and *action protocols*, which are transcripts of the speech and actions of participants as they perform the activity. Each transcript is elicited with the aid of a human prompter, who asks the participant to “think aloud” during the performance. Transcripts are analyzed by segmenting them and mapping their components into the concepts and operations of a given candidate task model. A *coding scheme* specifies how patterns of speech and actions are expected to correspond to the performance of specific tasks in the candidate

Code	Line	Protocol text
ProgComp:Read	52:	where is the dispatch method defined [searches for method in source]
FaultId:Exec	53:	let's add some printf's to see where this error is coming from [edits source]

Figure 1: Sample excerpt from an encoded protocol.

model. These coding schemes are developed *a priori* by domain experts and used by independent analysts (called *coders*) who apply them to raw protocols to yield *encoded protocols*.

For example, Fig. 1 depicts part of an encoded protocol. The rightmost column constitutes a transcript in which the bracketed text describes actions and the unbracketed text quotes the participant. The transcript has been partitioned into numbered segments (middle column), and each segment has been assigned a code (left-most column), which maps the segment to part of the model. In this case, the model (omitted due to space constraints) represents program comprehension and fault identification activities.

Encoded protocols are used to judge the fitness of a candidate model as follows. While many segments of a raw protocol may yield an encoding, other segments may fail to do so (e.g., observed activity that is not part of the model). In addition, sequences of encoded segments may either match or fail to match the ordering predicted by the model, and the model may predict encodings or sequences of encodings that are not observed in any raw protocol. Finally, when multiple models are being compared, one would use multiple coding schemes, one for each model, and use the relative fitness of the encoded protocols to choose the best model.

3. PROCEDURE

Our pilot study will involve graduate students in a formal-methods course at Michigan State University. Participants will perform maintenance activities on a multi-threaded browser program, which reads lines of text from the network and displays them in a graphical window. The program is small (roughly 4,500 LoC) and is implemented using a model-view-controller architecture, which is representative of a large class of GUI programs. The browser has two threads of control—a network manager, which reads data from the network, and a GUI manager, which handles GUI events.

The study will comprise two phases, each involving the performance of a maintenance activity. We designed these activities to require changes to the browser's synchronization logic, thus stimulating the performance of the tasks we are trying to model. In the first phase, participants, unfamiliar with the code base, will be asked to add a status bar that reports various network server and GUI information (e.g., transmission rate as reported by a server and current GUI window size). The second phase will use the same code base, thus providing an opportunity to study maintenance activities over familiar programs. Participants will be asked to alter the code so that old text is periodically cleared from the browser window. Prior to the first phase, participants will participate in several training sessions that cover prerequisite knowledge (e.g., the thread library interface and the architecture of the program).

The general procedure will be the same for both phases. Each participant will be isolated in a controlled environment. The participant will be given a workstation configured with common software development tools, Web access, a microphone, and video recording software, which will record the video feed to the monitor and the audio feed from the microphone. The activity begins when the participant is given the program artifacts and a brief English specification of the change. Participants will have three hours to complete each activity.

We will also collect auxiliary data about the participants and their knowledge and performance. Prior to each activity, we will admin-

ister a test covering understanding of general concurrency concepts and the design of the browser program. Following the second activity, a similar test will be administered. Finally, to better understand each participant's performance, we will show her the video and interview her as she watches to get her *a posteriori* interpretation.

4. DISCUSSION

We recognize several threats to the validity of any model we develop using this procedure. These threats include the use of students rather than practicing engineers and the limited scale of both the program and the change activities. We feel graduate students should suffice for a pilot study and will further validate the model using industrial practitioners in a future study. The scale problem is more difficult to address: Participants can only be asked to participate for a reasonable length of time, and the analysis of the protocols themselves is extremely time-consuming. We will address these threats in future work by repeating our study with a wider sampling of participants and artifacts. The use of the think-aloud method is a potential threat to validity because the cognitive resources required for introspection may affect how participants perform. Fortunately, numerous studies show that participants who are asked to vocalize their "inner speech" perform comparably on measures of performance with participants who are not asked to think aloud [3]. Ericsson provides guidelines on avoiding this threat.

A technical problem we will need to address with respect to our task model is how to handle variations in our participants' approaches. It may be possible to explain much of these variations in terms of individual differences, such as working memory, spatial visualization abilities, distractibility, classification skills, fluid intelligence, and learning-style preference. Standard tests exist to assess these differences. Such data could help us evaluate and better understand interactions between innate ability and approach to solution, and the effects of these factors on performance. Moreover, we will be able to reason about how well articulated our model is with respect to participants with different cognitive traits.

In addition to the task model, our pilot study should contribute a number of artifacts to the empirical software engineering community, including the coding schemes we develop and the transcripts of raw protocols. To further the evolution of the model, we encourage interested parties to replicate and participate in our studies.

5. REFERENCES

- [1] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.
- [2] L. Cousin and J. S. Collofello. A task-based approach to improving the software maintenance process. In *Proc. of Conference on Software Maintenance*, 1992.
- [3] K. A. Ericsson. Valid and non-reactive verbalization of thoughts during performance of tasks. *Journal of Consciousness Studies*, 10(9–10):1–19, 2003.
- [4] K. A. Ericsson and H. A. Simon. *Protocol analysis: verbal reports as data*. MIT Press, 1993.
- [5] M. Hanna. Maintenance burden begging for a remedy. *Datamation*, pages 53–63, 1993.
- [6] D. Lea. *Concurrent Programming in Java™: Design Principles and Patterns*. Addison-Wesley, second edition, 2000.
- [7] J. Magee and J. Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, 1999.
- [8] A. Newell and H. Simon. *Human Problem Solving*. Prentice Hall, 1972.
- [9] E. Newman, A. Greenhouse, and W. L. Scherlis. Annotation-based diagrams for shared-data concurrency. In *Workshop on Concurrency Issues in UML*, 2001.
- [10] J. Sillito, G. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2006.
- [11] E. Soloway, R. Lampert, S. Letovsky, D. Littman, and J. Pinto. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11), 1988.
- [12] M. W. van Someren, Y. F. Barnard, and J. A. C. Sandberg. *The Think Aloud Method*. Academic Press, London, 2004.