Parallel Program Comprehension

Eric Aubanel
Faculty of Computer Science
University of New Brunswick
Fredericton, New Brunswick
Canada, E3B 5A3
aubanel@unb.ca

Abstract

Parallel programming keeps growing in importance, driven both by changes in hardware and the increasing size of data sets. Hundreds of parallel languages have been proposed, but very few have taken hold beyond the language developers themselves. One reason for this is usability - that is the degree of ease with which one can develop and maintain parallel programs that are both correct and reach the desired level of performance. The few studies of parallel language usability have not been informed by a theoretical framework. Existing theoretical models of program comprehension need to be extended to parallel programming to help address the challenges of developing new languages, programming frameworks, development tools, and pedagogy. The contribution of this article is to motivate research on parallel program comprehension, and to suggest a way forward by expanding the two-level program/situation model of program comprehension to include a model of program execution and by applying the extensive work on human reasoning by Johnson-Laird to understand how people reason about parallel programs.

1. Introduction

The cognitive psychology of computer programming has been well studied since the 1970's, and has led to deep insight into how programmers design, build and understand software (Détienne, 2001). This knowledge is vital for the development of software engineering tools and techniques, for the design of programming languages, and for computer science education. Program comprehension is relevant to many programming tasks. When implementing a design, programmers need to read and assess what they have written. Design also frequently involves reuse, which requires comprehension of the code to be reused. Other programming tasks require program comprehension, such as modification to add features, improve performance or software quality, and of course debugging. In order to comprehend a program, a programmer constructs a mental representation based on the program text and the programmer's knowledge (Détienne, 2001).

Theories about the mental representation of computer programs have informed the research and development of software engineering tools (Storey, 2006). Existing theoretical models of program comprehension include knowledge stored in long term memory (language syntax and semantics, programming schemas) and the development of mental models of programs in working memory. These components were brought together in von Mayrhauser and Vans's Integrated Code Comprehension Metamodel (1994).

The mental model theories of program comprehension are based on theories of natural language text comprehension (Détienne, 2001). One important question is whether a text is represented mentally by its propositional structure or by its meaning. Johnson-Laird has made a strong case that it is the meaning that is represented, in the form of mental models (P. N. Johnson-Laird, 1983). For computer program comprehension the propositional structure is referred to as the program model and the meaning is represented by the situation model. This two-part model has been successfully applied to the comprehension of procedural and object-oriented programs (Détienne, 2001), but has not been studied for parallel programs. Understanding a parallel program requires additional work, such as reasoning about multiple streams of execution and awareness of execution at the machine level.

Parallel programming keeps growing in importance, driven both by changes in hardware and the increasing size of data sets (Asanovic et al., 2006). Hundreds of parallel languages have been proposed, but very few have taken hold beyond the language developers themselves. One reason for this is usability - that is the degree of ease with which one can develop and maintain parallel programs that are both correct and reach the desired level of performance. The few studies of parallel language usability have not been informed by a theoretical framework (Mattson & Wrinn, 2008). Sadowski and Shewmaker's (2010) survey found that the existing literature on usability of parallel programming languages was inconclusive and that there were significant challenges in measuring usability.

The existing theoretical models of program comprehension need to be extended to parallel programming to help address the challenges of developing new languages, programming frameworks, development tools, and pedagogy. The contribution of this article is to motivate research on parallel program comprehension, and to suggest a way forward by expanding the two-level model of program comprehension to include a model of program execution and by applying the extensive work on human reasoning by Johnson-Laird to reasoning about parallel programs.

Sections 2-4 review three types of mental models. Section 2 reviews the mental model theory of program comprehension, and concludes with an illustrative example to discuss the extra work required in parallel program comprehension and introduce the idea of an execution model. Section 3 discusses the importance of machine models in parallel programming, and how they are related to the concept of notional machines. Section 4 briefly presents Johnson-Laird's understanding of human reasoning with mental models. Finally, Section 5 presents a proposal for an execution model component of program comprehension and how it might be used in reasoning about parallel programs.

2. Mental Models in Comprehension of Computer Programs

The experimental study of computer program comprehension dates back to the early '80s (Bidlake, Aubanel, & Voyer, 2020). Détienne provides a thorough analysis of work up until the late '90s in her book Software Design – Cognitive Aspects (2001). She classifies work on program comprehension into approaches that use schemas, representing domain and programming knowledge, problem solving approaches, and the mental model approach. According to Détienne (2001, ch. 6), "It appears that the mental model approach is the one that explains most completely the processes employed and the representations constructed in the course of understanding a program."

The mental model approach began with work by Pennington (1987). Pennington's experiments revealed that programmers construct a program model using control flow structures, when reading a program for the purpose of comprehension. When the comprehension stage is followed by a modification stage, which requires comprehension of the meaning of the program, a situation model is constructed. The situation model represents the program's data flow and goals. Any competent programmer can construct the program model. The situation model is constructed when the meaning of the program is important, and is more difficult to construct than the program model. The data flow is not as obvious as the control flow, and the function of the program is the hardest to discover.

Later work expanded Pennington's model by considering the effect of expertise, programming paradigm, and task (Bidlake et al., 2020). While both novice and expert programmers show no differences in the construction of the program model, novices do have more difficulty in constructing the situation model (Burkhardt, Détienne, & Wiedenbeck, 2002). Object-oriented program comprehension does not proceed in the same way as procedural program comprehension, in that both program and situation models are constructed in parallel. Burkhardt et al. expanded the program model to include a macrostructure consisting of the control flow between functions. They expanded the situation model for OO programs to take into account objects and their interrelationships. The majority of program comprehension studies use program understanding, also known as read-to-recall as their task (Bidlake et al., 2020). The read-to-recall task is to remember program code after a study period, either by answering questions about the code or paraphrasing it. Other tasks used in studies can be classified as read-to-do, which includes modification, debugging, and classifying programs (Bidlake et al., 2020). As suggested in Pennington's study and confirmed in later work, the development of the situation model is more likely given a read-to-do task, where understanding the meaning of the program is important.

2.1. Illustrative Example

There are many parallel programming models, suitable for parallel execution using vector instructions and threads on multicore processors and graphics processing units, and processes across processors in a cluster. We use OpenMP as an easy to understand and popular shared memory programming model in our illustrative examples. Consider the nested loops in Figure 1 written in the C programming language, annotated with an OpenMP parallel directive (Liao, Lin, Asplund, Schordan, & Karlin, 2017).

```
double a[len][len];
\\ ...
#pragma omp parallel for private(j)
for (i = 0; i < len - 1; i += 1) {
  for (j = 0; j < len ; j += 1) {
    a[i][j] += a[i + 1][j];
  }
}
```

Figure 1 – C/OpenMP simple race condition example

We can examine this code as a miniature program comprehension exercise, and identify the components of the two-level program/situation model. We'll start by ignoring the OpenMP `pragma`. The program model contains both a micro- and a macro-structure (Détienne, 2001), but here only the former is relevant. The microstructure represents the surface details and the control flow of the program: two nested `for` loops updating elements of a two-dimensional array `a`. This model is built automatically by any programmer with syntactic/semantic knowledge of the language. The situation model has static and dynamic components. Here the dynamic situation model represents the data flow of the program and the static situation model represents the goal of the program. Construction of the situation model is optional, and takes more effort. It involves tracing updates to the matrix, where elements of each row are replaced by the sum of their value and their lower neighbour. In this small code sample this is also the goal of the program.

The OpenMP `pragma` tells the compiler to parallelize the outer loop by forking threads and assigning contiguous blocks of iterations to them. All variables are shared among threads by default, except for the iteration counter `i` (implicitly) and the counter `j` (using the `private` clause) of the inner loop, which are private. This adds to the program model the text itself, but not its meaning, other than its identification as a compiler directive. It adds to the situation model the parallelization of the outer loop and the knowledge that the iteration variables `i` and `j` are private to each thread, that `a` is shared, and that there is an implicit barrier at the end of the outer loop. Analysis of the data flow must now take into account multiple threads. This analysis reveals a problem, namely a data race. A thread working on row `i` could read from a value in row `i+1` while a thread working on row `i+1` is writing to the same memory location, leading to incorrect results.

Whereas the above analysis of the comprehension of the non-parallel code is based on mental structures for which there is considerable experimental evidence, there is no evidence that the analysis in the previous paragraph reflects how programmers think about parallel programs.

Full parallel program comprehension might even seem impossible. In the words of Skillicorn and Talia (1998), "An executing parallel program is an extremely complex object." There may be hundreds of threads executing concurrently, and threads may communicate with each other synchronously or asynchronously. The interleaving of memory accesses by multiple threads may change from execution to execution, which can lead to nondeterministic results in a faulty program. How can comprehension of such complex execution happen? We propose that the programmer builds mental models of representative cases of parallel execution, and then reasons about the correctness and meaning of the code using these models. We further propose to introduce a new component to the memory model theory, namely the execution model, which represents the execution of a program. In the example above, the execution model would represent the execution of multiple threads and how they lead to a data race by reading and writing to the same locations of the `a` array.

3. Notional Machines and Machine Models

Texts written in a high-level programming language must be translated into machine instructions in order to be executed on a computer. This has important implications for the programmer's mental model. Any educated programmer knows that a single high-level instruction may be translated into multiple low-level instructions. However, as the programmer traces through some code and executes it in their working memory, they do so on an abstract mental representation of a machine that can execute the high-level instructions directly. This abstract machine has been called the **notional machine** in the context of computer science education. (Du Boulay, 1986): "A notional machine is a characterization of the computer in its role as executor of programs in a particular language or a set of related languages." (Sorva, 2013, p. 2). In this definition 'computer' refers to both hardware and system software (compiler/interpreter, operating system). There can be multiple notional machines for the same language, at different levels of abstraction. For example, it's possible to mentally execute a C program without considering how memory is divided into stack and heap. It's also possible to track the memory management with a lower-level notional machine.

Notional machines, together with the literature on knowledge mental models, are valuable for computer science pedagogy. Experiments have shown that novice programmers' mental models are inadequate (Sorva, 2013), and computer science instructors commonly observe students' superstitions about the behaviour of the notional machine. A challenge in educating programmers is to help them build viable notional machines, so that they can accurately reason about programs and simulate them mentally.

We argue that something akin to notional machines is relevant for expert program comprehension. This knowledge can be called a machine model. It adds the cost of execution to the programming schema knowledge that experts possess, including the contribution of compilers, operating systems, runtime software, and hardware. For instance, it allows programmers to assess the overhead of function execution and the desirability of function inlining. For a programmer performing incremental parallelization of a sequential program, knowledge of the machine model allows them to reason about whether parallelizing a loop is worthwhile, based on the cost of the parallelization overhead. The machine model also supports the dynamic aspects of the mental model of the expert programmer, when faced with a comprehension task. While comprehension relies to a large extent on static knowledge of programming plans, it also can involve the dynamic aspects of the mental model, particularly data flow. Mental execution may be required if the programmer is unfamiliar with a programming plan, either because the plan doesn't follow the rules of programming discourse or because the programmer has not seen the plan before (Détienne, 1990). For parallel programming, reasoning about the execution of the program is crucial in assessing correctness and performance, as in the example in Figure 1.

Previous work has not emphasized the dynamic aspects of program comprehension. We argue that it would be fruitful to move comprehension of data flow from the situation model into a separate component called the execution model. We believe that the construction of the execution model in working memory depends on the machine model that is used, but we will focus on the former in what follows.

4. Mental Models for Reasoning

According to Johnson-Laird there are at least three types of mental representations:

1. Propositional representations (strings in a natural language)
2. Images, which are perceptions from a particular point of view
3. Mental models, "which are structural analogues of the world" (P. N. Johnson-Laird, 1983)

Mental models can be manipulated, and can be used to reason without formal logic. The structural analogues are usually two or three-dimensional icons. As Johnson-Laird explains in a 2010 review, "A visual image is iconic, but icons can also represent states of affairs that cannot be visualized", such as "the abstract relations between sets that we all represent." (P. N. Johnson-Laird, 2010). These icons are not restricted to a single point of view but can be manipulated.

Johnson-Laird's main concern is with understanding how people reason using mental models. His research has convincingly demonstrated that humans don't reason using formal logic (the "doctrine of mental logic"). He makes an important point about the complexity of reasoning, which is relevant to the comprehension of parallel programs: "Almost all sorts of reasoning,..., are computationally intractable. As the number of distinct elementary propositions in inferences increases, reasoning soon demands a processing capacity exceeding any finite computational device,..., including the human brain" (P. N. Johnson-Laird, 2010). Humans deal with this complexity by constructing representative mental models.

Consider the following sentence (P. N. Johnson-Laird, 2004):

> The cup is on the right of the spoon

It might be reasonable to postulate that the meaning of this sentence is represented by the reader in a mental language. What if we add three more sentences:

> The plate is on the left of the spoon.
> The knife is in front of the cup.
> The fork is in front of the plate.

and ask the question: what is the relation between the fork and the knife? Answering this question requires spatial reasoning, not reasoning about language. A mental model for these four premises can be given as:

```
> plate spoon cup
> fork        knife
```

This model can then be used to give the answer to the question: the fork is on the left of the knife.

Consider now the same four premises with a small change to the second one:

> The plate is on the left of the cup.

These premises can be represented with at least two models, the model above and this one:

```
> spoon plate cup
>       fork  knife
```

The answer to the question is still the same (the fork is on the left of the knife), however the reasoning is more difficult, because more than one model needs to be considered. The extensive literature on reasoning with mental models is likely to contain insights into how programmers reason about code.

5. Execution Model

Comprehension of the illustrative example in Figure 1 requires applying knowledge of how the loop iterations are assigned to threads (the C/OpenMP machine model). Comprehension of the parallel aspect of this code requires reasoning about the data access pattern of the threads, in other words the parallel data flow. This comprehension can be done independently of the understanding of the meaning of the program, which forms the static part of the situation model. The programmer, faced with the task of verifying the correctness of this code, could conceivably focus on the parallel data flow without bothering to understand the meaning of the code. In contrast, the essence of the situation model in text understanding is the situation of the text, that is its meaning.

We propose to carve out a separate mental execution model, which would include the dynamic aspect of the situation model, namely the data flow. The separation of the execution model could account for the separate understanding of the behaviour of the the program from its meaning. This behaviour occurs at two levels: data flow in the program text and data flow in the computer. The latter behaviour is key to understanding parallel programs. It is also key to understanding the performance of any program, for example the flow of data between levels of the memory hierarchy. The first level bears more discussion. While the data flow of a program can be considered between variables in a program, it is often considered in the context of one or more data structures. Part of the comprehension of a non-trivial program is understanding the data structures, which are not evident in the surface of the text. They form part of the understanding of the underlying algorithms used in the program. Data structures do not seem to have been considered in models of program understanding (Détienne, 2001, ch. 6, footnote p. 94).

The execution model can be divided into three layers of abstraction. At the top, it represents the behaviour of the operations on the data structures. In the middle, the data flow between variables in the program text. At the bottom, the data flow in the processing elements of the computer.

The comprehension model in Table 1 modifies Burkhardt et al.'s version of the program/situation model (2002), which accounts for object oriented programs, by moving data flow from the situation model to the execution model, and adding two aspects of data structures to the execution and situation models. The identification of the data structure refers to the program's meaning, and is part of the situation model. The behaviour of the data structure, that is its mutation by one or more threads of execution, is part of the execution model. The parts of the execution model that are unique to parallel programs are discussed in the following example.

Program Model	Execution Model	Situation Model
Microstructure:	Data structures (behaviour)	Data structures (identification)
Program statements	Data flow (text)	Program goals
Control flow	Data flow (machine)	Domain objects (classes)
Macrostructure:	**Parallel programs:**	Relations between objects
Functional structure	Decomposition	Communication between objects
Control flow between functions	Communication	

Table 1 – Three-part comprehension model.

5.1. Another Illustrative Example

Consider another C/OpenMP example (Mattson & Meadows, 2014) in Figure 2, where `p` is a struct containing a `node *next` pointer and a pointer to some payload that will be used in the execution of the `process()` function.

Ignoring the parallel directives for the moment, the following comprehension process can be sketched. The program model would identify a loop that continues as long as pointer `p` is not `NULL`, the initialization of pointer `p` and its updating in the loop, and its passing to function `process`. The situation model would contain the meaning of this code as the processing of data that is stored in a linked list. The role of the execution model could depend on whether the programmer is a novice or an expert. An expert would not likely have to mentally invoke the dynamic aspect of the linked list; its static meaning should be sufficient. A novice who was unfamiliar with linked lists, might need to trace execution of the linked list, that is invoke the execution model, on their way to understanding its meaning.

The execution model is more pertinent to the parallel version of the code. Adding parallel directives doesn't change the meaning of the code, so the situation model is unchanged. It does require understanding how the code executes in parallel, hence the execution model is vital. This understanding

```
#pragma omp parallel{
 #pragma omp single{
  node * p = head;
  while (p) {
   #pragma omp task
   process(p);
   p = p->next;
  }
 }
}
```

Figure 2 – C/OpenMP parallel linked list example

includes the forking of threads at the beginning of the code, followed by the restriction of the code's execution to a single thread. Each iteration of the `while` loop involves creation of a new OpenMP task which is scheduled by the OpenMP runtime on another thread. In other words, one thread dispatches the tasks, which are executed by multiple threads.

Decomposition and communication are two key parts of the execution model which are found only in comprehension of parallel programs. The former is an essential part of any parallel program (Aubanel, 2016). The example in Figure 2 exhibits a trivial decomposition into tasks. Data decomposition also features prominently in parallel computing. The example of Figure 1 exhibits decomposition of the two-dimensional array into blocks of rows. While commmunication is not explicit in shared memory programs such as these, it can be present implicitly in the ordering of memory accesses of the threads. Communication should form an explicit part of the execution model of distributed memory programs, such as those that involve message passing.

5.2. Reasoning with the Execution Model

A mental model represents a possibility (P. Johnson-Laird, 2001). What are the possibilities for the execution model for the example in Figure 1? Knowledge of the semantics of the OpenMP `parallel for` directive reveal that the default scheduling is to partition the the iterations of the outer loop into contiguous blocks, one per thread. The execution model would represent the decomposition of the 2D array into blocks of rows and the dependence between blocks arising from `a[i][j] += a[i + 1][j];`. It also needs to represent the dynamic behaviour of the threads. A simplifying (but not generally correct) assumption is that threads start at exactly the same time and proceed in lock step. This means that a thread working on the last row of its block would use values in the next row which had already been updated by the thread working on the next block, yielding incorrect results. This single possibility can be used to identify the data race.

Programmers can't possibly follow the execution of multiple threads, especially since the relative timing of the threads can vary from one execution to the next. Instead, programmers construct mental models for representative cases, each corresponding to an interleaving of a particular number of threads. Reasoning gets more difficult as the number of mental models increases (P. Johnson-Laird, 2001). The code in Figure 3 (Liao et al., 2017) represents a more subtle race condition, where `xa1` and `xa3` point to two elements of an array, where `xa3 - xa1 = 12`.

As hinted by the note, there is a dependence between iterations 0 and 5, where `xa1[521]` and `xa3[533]` point to the same location. This is not a problem if iterations 0 and 5 are handled by the same thread; a race condition occurs if they are handled by different threads. Assuming the same static scheduling of the loop iterations, this means that iterations 0 and 5 would be handled by the same thread if the total number of threads is less than 36. This requires mental models for two possibilities: number of threads < 36 and ≥ 36. The programmer could miss the race condition by using a single possibility of less than 36 threads, which could easily happen if they are used to working with a small number of threads.

```
#define N 180
int indexSet [N] = {
//Note: indexSet[5] - indexSet[0] = 533-521= 12
  521, 523, 525, 527, 529, 533,
  547, 549, 551, 553, 555, 557,
// omitted code here ...
};
#pragma omp parallel for
for(i=0; i< N; ++i){
   int idx=indexSet[i];
   xa1[idx]+=1.0;
   xa3[idx]+=3.0;
}
```

Figure 3 – C/OpenMP tricky race condition example

6. Acknowledgements

The author acknowledges the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), RGPIN-2018-04811.

7. Conclusion

Understanding a program involves more than determining its meaning. It also involves understanding its dynamic behaviour. This is particularly important for the comprehension of parallel programs, where the behaviour of multiple streams of execution must be understood. We have argued that adding an execution model to the current mental model theory would account for this understanding. This includes data structures and their decomposition in the mental representations of parallel programmers.

Program comprehension involves knowledge stored in long term memory in addition to the mental representations created in working memory during comprehension. Understanding a parallel program requires a mental model of the parallel system, which could be viewed as a notional machine for expert programmers. Different parallel programming languages have different machine models (Aubanel, 2016; Skillicorn & Talia, 1998). The impact of a language's machine model on the comprehension of its execution would be worth studying. Languages that have a high level of abstraction may make it difficult to understand what is happening at the machine level, which is necessary in order to reason about performance. Languages at a lower level of abstraction may involve a tradeoff between exposing execution at the machine level and increasing the cognitive load of having to understand multiple streams of execution.

How do programmers actually reason about a parallel program's behaviour? They likely use their knowledge about the programming language's machine model to construct representative cases. This could be similar to how people reason about natural language texts, and could lead the way to understanding what makes one program harder to understand than another, and what kind of mistakes even expert programmers make. Understanding how parallel programmers reason, and the challenges they face, could lead to the development of representations that would aid in comprehension. This could include diagrammatic representations, to show the decomposition of data structures and the communication between tasks. Without knowing anything about programmers' mental representations, it's hard to predict whether a proposed representation would be helpful.

8. References

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., ... others (2006). *The landscape of parallel computing research: A view from Berkeley* (Tech. Rep.). Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Aubanel, E. (2016). *Elements of parallel computing*. CRC Press.

Bidlake, L., Aubanel, E., & Voyer, D. (2020, July). Systematic literature review of empirical studies on mental representations of programs. *Journal of Systems and Software*, *165*(110565).

Burkhardt, J.-M., Détienne, F., & Wiedenbeck, S. (2002). Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, *7*(2), 115–156.

Du Boulay, B. (1986, February). Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, *2*(1), 57–73.

Détienne, F. (1990). Expert Programming Knowledge: A Schema-based Approach. In *Psychology of Programming* (pp. 205–222). Elsevier.

Détienne, F. (2001). *Software Design–Cognitive Aspects*. Springer Science & Business Media.

Johnson-Laird, P. (2001). Reasoning with Mental Models. In *International Encyclopedia of the Social & Behavioral Sciences* (pp. 12821–12824). Elsevier.

Johnson-Laird, P. N. (1983). *Mental Models*. Cambridge University Press.

Johnson-Laird, P. N. (2004). The history of mental models. In *Psychology of reasoning* (pp. 189–222). Psychology Press.

Johnson-Laird, P. N. (2010, October). Mental models and human reasoning. *Proceedings of the National Academy of Sciences*, *107*(43), 18243–18250.

Liao, C., Lin, P.-H., Asplund, J., Schordan, M., & Karlin, I. (2017). DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '17* (pp. 1–14). Denver, Colorado: ACM Press.

Mattson, T., & Meadows, L. (2014). *A "Hands-on" Introduction to OpenMP*. Retrieved 2020-03-25, from `https://extremecomputingtraining.anl.gov/files/2016/08/Mattson_830aug3_HandsOnIntro.pdf`

Mattson, T., & Wrinn, M. (2008). Parallel programming: can we PLEASE get it right this time? In *Proceedings of the 45th annual Design Automation Conference* (pp. 7–11). ACM.

Pennington, N. (1987). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, *19*(3), 295–341.

Sadowski, C., & Shewmaker, A. (2010). The last mile: parallel programming and usability. In *Proceedings of the FSE/SDP workshop on Future of software engineering research* (pp. 309–314). ACM.

Skillicorn, D. B., & Talia, D. (1998, June). Models and languages for parallel computation. *ACM Computing Surveys (CSUR)*, *30*(2), 123–169.

Sorva, J. (2013, June). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, *13*(2), 1–31.

Storey, M.-A. (2006, September). Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, *14*(3), 187–208.

von Mayrhauser, A., & Vans, A. M. (1994). Comprehension Processes During Large Scale Maintenance. In *Proceedings of the 16th International Conference on Software Engineering* (pp. 39–48). Los Alamitos, CA, USA: IEEE Computer Society Press.