Identifying Domain-Specific Defect Classes Using Inspections and Change History

Taiga Nakamura
University of Maryland
4122 A.V. Williams Building
College Park, Maryland USA
1-301-405-2721
nakamura@cs.umd.edu

Lorin Hochstein
University of Maryland
4122 A.V. Williams Building
College Park, Maryland USA
1-301-405-2721
lorin@cs.umd.edu

Victor R. Basili
University of Maryland
4111 A.V. Williams Building
College Park, Maryland USA
1-301-405-2668
basili@cs.umd.edu

ABSTRACT
We present an iterative, reading-based methodology for analyzing defects in source code when change history is available. Our bottom-up approach can be applied to build knowledge of recurring defects in a specific domain, even if other sources of defect data such as defect reports and change requests are unavailable, incomplete or at the wrong level of abstraction for the purposes of the defect analysis. After defining the methodology, we present the results of an empirical study where our method was applied to analyze defects in parallel programs which use the MPI (Message Passing Interface) library to express parallelism. This library is often used in the domain of high performance computing, where there is much discussion but little empirical data about the frequency and severity of defect types. Preliminary results indicate the methodology is feasible and can provide insights into the nature of real defects. We present the results, derived hypothesis, and lessons learned.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging – *code inspection and walk-throughs, debugging aids, testing tools.*

General Terms
Reliability, Experimentation, Human Factors, Languages.

Keywords
Domain specific defects, code reading, inspection, change history.

1. INTRODUCTION
Since debugging is one of the most time-consuming tasks in software development, the software engineering community is very interested in understanding software defects. Although there have been numerous achievements in this area, many interesting research questions on defects are still hard to answer due to an absence of empirical data. For example, organizations may want to know what kinds of defects are frequently made in their software projects, so that they can understand where to focus their efforts during the testing and verification phase. In addition, developers who start using a new language want to know what kinds of defects tend to be difficult to deal with, so that they can understand the strengths and weaknesses of the language. The reason these questions are difficult is that the answers depend on specific contexts in which the software was written in a particular domain, so the results obtained in one domain are often not applicable to other domain. To address these kinds of problems, empirical research is needed.

Previous software engineering research on defects has often been based on the analysis of defect reports or change requests, which are created by testers and users in various forms [2][20]. Unfortunately, such artifacts are often either incomplete or not available at all in many software projects. Even when such reports are available, they seldom contain the appropriate context information for classifying defects in a useful way. Another common approach is to analyze metrics which are believed to be correlated with defects (e.g. job steps [3], program changes [8]). However, such metrics provide little insight into the nature of the actual defects.

In this paper, we present a methodology of defect analysis that uses existing software with source code history. It can be used to construct a defect classification scheme as well as a profile of the frequency and severity of the different types of defects that have occurred. One advantage of our methodology is that this allows us to identify domain-specific defects by examining the actual defects that exist in the code. Our approach is "reading-based," meaning that our defect analysis is driven by a human reading the code, identifying defects and classifying them. The results are iteratively refined through various verification methods. Unlike other reading techniques, this method involves examining multiple versions of the source code, and assumes the existence of a source code repository. This repository allows us to capture the defects that existed in intermediate versions but were found and fixed in the final code. We claim that the methodology we present is effective and powerful if it is appropriately designed and executed.

Throughout this paper, we use the results of the case study conducted in the domain of High Performance Computing (HPC) to present concrete examples for the methodology. Development in HPC is distinctive from generic software development in several aspects:

- Powerful computation power required in today's HPC systems is achieved by hardware with massively parallel processing. For example, the world's fastest supercomputer as of writing this paper consists of 131,072 processors [18]. HPC software needs to be written so that it can scale up well with the number of processors and the size of data.
- To leverage this parallelism, developers usually use special HPC languages such as MPI (Message Passing Interface) [6], OpenMP [5], and CAF (Co-Array Fortran) [14] associated with Fortran, C, etc. In addition, various new HPC languages are being developed.
- HPC systems are often developed by scientists and students who have not had formal training in software engineering. Very few traditional software engineering processes or practices are used in HPC projects.
- Traditional command-line tools and programming styles are more commonly used than modern GUI tools or object oriented design.
- Emphasis is put on both correctness and performance. That is, an HPC program can contain performance defects even if it produces correct output.

While many HPC projects use a source code management system, the use of a defect tracking system is very limited. Therefore, our method is well-suited for establishing knowledge on defects specific to the HPC domain.

The organization of this paper is as follows. After reviewing related work in Section 2, we describe our defect analysis methodology in Section 3. In Section 4 and 5, we describe the methodology of analysis and verification with the results of case study. In Section 6, we discuss tool support. We present discussions in Section 7, and we present a summary in Section 8.

2. RELATED WORK

Code inspections are a well-known family of methods for identifying defects. Many kinds of inspection techniques have been developed with differences in how each method is applied and which technique is used to detect defects [7]. The goal of code inspection is to improve software quality by finding defects. Although our method is very similar to conventional code inspection, the goal of our analysis is to characterize and build knowledge of recurring defects rather than just identifying the defects left at the time of the inspection so they can be fixed. While a normal code inspection usually examines the code at a specific point of time, our analysis tries to identify defects that existed in intermediate versions. By examining how much time was spent on finding and fixing these defects, we can determine which defect types are more important.

More recently, the research community has become more interested in mining source code history to obtain useful insights. In [21], information obtained from code history is used to enhance the accuracy of the defect detection tool.

The Marmoset system [16] is an automatic source code collection and testing tool built on top of the Eclipse platform. Like the data collection system we use, Marmoset captures the source code as the subjects work on class assignments. Although the collected data is potentially applicable to our methodology, their focus is on providing quick feedback to both students and TAs by the automated testing mechanism.

Our approach is an attempt to formulate and accelerate the activities done by researchers of defect finding tools and techniques. When researchers plan to develop a new technology, they often examine the source code as a preliminary study to guess what kind of defects are worth preventing and identifying. [9][21] By doing so, they can be more confident that their technology will solve a real problem rather than a hypothetical problem. While they sometimes perform extensive reading-based analysis, they do not focus on accumulating the knowledge as we have done, since their goal is to get hints for technology development.

3. OVERVIEW OF THE METHODOLOGY OF DEFECT ANALYSYS

In this section, we present the overview of our analysis methodology. Figure 1 illustrates the methodology we propose. The process of defect analysis consists of the following main activities.

- **Support**: Supplemental activities to assist analysis
- **Analysis**: Analyze the code, record the identified records, and develop classification scheme and hypotheses.
- **Verification**: Verify the analysis results at various levels

As shown in Figure 1, there are feedback loops to make all of these activities iterative as more data are analyzed and feedback is given by the verification process.

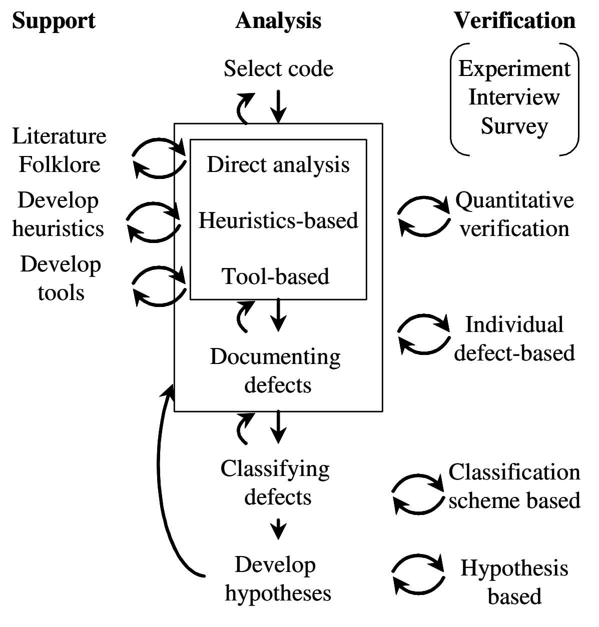

Figure 1. Methodology of defect analysis.

In the later sections, we present each activity of the methodology and its concrete realization in the case study in the HPC domain interleaved, so that we can discuss both the methodology itself and the issues when it is applied to a specific domain.

4. DEFECT ANALYSIS

In this section, we describe the methodology of analysis. Our approach is conducted bottom-up, i.e., building high-level knowledge by synthesizing and abstracting low-level findings. The analysis is conducted as a series of the following activities: Selecting code to analyze, analyzing the code, documenting identified defects, classifying defects and developing hypotheses. Since it is difficult to complete all activities in a single pass, we use an iterative approach. Iteration occurs at several different levels between and across steps.

Below, the methodology for each activity, and how it was applied to the case study is described in detail.

4.1 Selection of Code to Examine

4.1.1 Methodology

The methodology requires that multiple versions of the source code are available for the analyst to identify defects as the code is changed over time. Depending on the target of the analysis, the code could come from one particular project, or it could come from a number of them. In either case there is some flexibility, as a project often consists of multiple source files. Furthermore, since we use data with a change history, multiple versions may be available for a single file, which increases the number of selections even more. The efficiency of the analysis is affected by the order in which the data are analyzed, because the analysis becomes easier as an analyst gains more knowledge about recurring defects. Also, when there is too much data, the analyst must prioritize them to make the analysis as effective as possible. Although we can never know in advance which code is worth examining, there are several strategies that can be used to select code.

Simple first: The analyst should begin with code that is already familiar. Before looking for defects, the analyst should understand aspects of the program such as code structure, algorithms and programming style. If there is no such prior knowledge, we recommend beginning with code that is easiest to understand. For example, smaller code (e.g. smaller files, earlier versions) tends to be easier to read. As analysts become more experienced with the methodology in the target domain, they can move to larger code which may contain more complex defects.

Pruning versions: When examining the code with the change history, the number of the versions of each source file can be an order of hundreds or thousands. Although our basic approach is to look at all versions of the source files exhaustively, it is often necessary to guess which version seems more important to shrink the search space. One approach is to identify the versions of files where large changes have occurred. Locating these files can help analysts focus their initial efforts on understanding how the code has evolved over time, without inspecting every single version manually. Large changes can be identified by measures such as total number of lines added and deleted. Figure 2 illustrates the size of changes in a hypothetical file change history. With the above strategy, the versions shown in parentheses can be skipped. For example, a large change from version 2 and 3 might contain function additions as well as new defects associated with them, so examining them can help understand what's happening. In addition, a series of small changes from version 3 and 6 might contain fixes to the defects made in version 3, so comparing 3 and 6 can help identifying defects. Therefore, 4 and 5 might be excluded from the initial inspection. Once defect-like code is located, the changes between individual versions should be examined more closely. The search space should be expanded gradually in the iterative process to capture missed defects.

If the initially inspected code is too difficult to analyze, it should be marked as "skipped" and other code should be examined. Since the code analysis is iterative, by examining the same code multiple times, more defects can be identified.

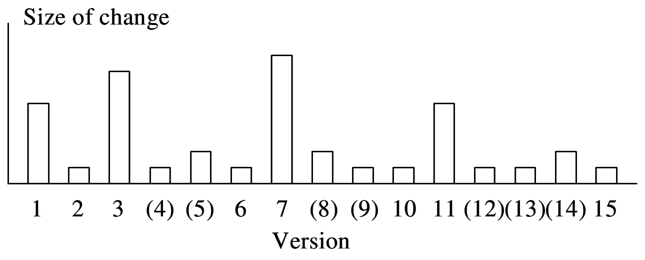

Figure 2. Size of changes in a hypothetical change history

External information: While the availability of the information from external sources such as project developers or a defect tracking system is limited in the situation we are interested in, if some of the defects are known to exist in particular versions of the code, the analyst can use that information reading these versions to confirm them with the lower cost.

4.1.2 Case study in HPC

In the case study, we have collected data from students solving small parallel programming problems as part of their class assignments. We captured a snapshot of the source code every time the students compiled their code on the parallel machine.

The source code analyzed for the case study was collected in 5 assignments. The data from 38 students have been manually analyzed and the identified defects have been recorded by the first author. All code analyzed was written in MPI with C. The number of code snapshots (i.e., the number of captured compiles) for each student's code varied from 5 to more than 500. It is unclear whether a very small number of code snapshots indicate that the student only needed to compile the program a few times, or that the student did most of the programming outside of the data collection environment.

The initial analysis focused only on C/C++ code that uses the MPI library. While we have also collected data from other parallel programming approaches, we decided to test the methodology by focusing our initial efforts on MPI, which is currently the most popular parallel programming model in HPC.

The number of versions we examined depended upon the size of the change history. When the number of versions was relatively small (<50), we examined all versions available. To reduce the number of versions examined in larger history data, we used the pruning strategy described in Section 4.1.1, using a "diff" tool to display the list of versions with the number of lines added and deleted. We found that our strategy for selecting pairs of versions to examine changed over time. Initially, comparing distant pairs of versions (e.g. versions 4 and 8 in Figure 2) did not yield much insight when searching for defects because it was difficult to understand all of the different kinds of changes mixed together. However, once we identified a defect and tried to examine how the subject fixed it, it made sense to look at the diffs across large changes because we could focus on a specific portion of the source code.

4.2 Code Reading

4.2.1 Methodology

The main part of the analysis is to examine the code and find defects. As already mentioned, our method is based on code inspection. As the idea of code inspection indicates, a human could be the most powerful defect detector, as the problem of determining whether a given code has a defect is not decidable in the general case. However, the analyst can easily get lost in volumes of code. Therefore, it is important to make the details of the analysis process as explicit as possible.

4.2.1.1 Direct manual analysis

When the analyst has little prior knowledge about the nature of the defects in the code, generic strategies can be applied that do not require domain-specific knowledge. The following are generic strategies for identifying defects.

- **Familiarize yourself with the code**: Look at a particular version of the source code to understand the code structure, the algorithm used to solve the problem, communication pattern, language features used, naming conventions for variables/functions, coding styles (e.g., many small functions vs. a few large functions, OO-like vs. procedural). It's often useful to read the latest (final) version, as it is the end result of the development efforts. (All intermediate versions converge toward this version.) It is also useful to read the early versions too, as the code tends to be small and simple, and thus easier to understand attributes such as programming style and initial structure.
- **Look at changes**: Examine a “diff” between particular versions and examine what has changed. For example, look at the diff between an intermediate version and the final version. The differences may contain bug fixes, debugging attempts, function additions, code refactoring, etc., but looking at how the code has been changed helps understand the defects the subject found and fixed. Note, however, it is necessary to look at the unchanged parts of the code as well, because (1) the defects may exist in the parts that have not been changed at all, as some defects may not have been noticed by the subject, and (2) the defects may exist in the global logic, instead of being localized in the region of the fix.
- **Look for specific defects**: Gather information about the kinds of defects that would be expected to occur in the software domain under investigation. Such information can be gathered from the literature, or by collecting folklore about defects from experienced developers in the appropriate domain. Collecting this knowledge either qualitatively or quantitatively by various empirical methods can contribute to enhancing an analyst’s ability to identify defects. The knowledge may take forms such as known defect types or classification schemes.

4.2.1.2 Heuristic-based detector analysis

As the analysis progresses and we obtain actual defect samples, it is possible that the analysts can define domain-specific heuristics to help find some defects. As discussed previously, the ability of analysts to find defects based on their own knowledge and experience accumulated is also a kind of heuristic. Unlike such “tacit” heuristics, however, the heuristics here should be explicitly documented (e.g., as a checklist), so that other analysts can also make use of this knowledge. For example, a heuristic might look like “if the function A is called with the parameter X in one processor, check if the function B is called with the parameter Y in the other processor.” Not all heuristics have to describe absolute defects, so including information on the confidence level is useful when possible.

Explicit heuristics can also be defined by information from developers. Again, the difference from the previous subsection is tacit versus explicit. The process of developing an appropriate set of heuristics is iterative. The heuristics should be refined as they are applied to more data.

4.2.1.3 Automated tool-based analysis

The heuristic-based approach described in the previous subsection is still a manual approach. To accelerate the analysis, it is important to automate the analysis process as much as possible. For example, if there is an existing tool that can identify a certain type of defect, applying it saves the cost of manual analysis for that defect. Under the assumption that there is no well-established knowledge on all defects that exist in the target projects, analysis with existing tools cannot completely replace manual analysis. However, since the methodology is iterative, new tools can be developed that detect recurring defects based on analysis from previous iterations. The use and development of the tools will be discussed in Section 6.

4.2.2 Defect analysis in the HPC case study

In the case study, to execute the analysis, we have used the following tools and techniques. The analysis has been done by one of the authors.

To efficiently examine the code, it is important to be able to access individual versions of the code snapshots systematically. In addition, we found that the ability to look at the differences between versions often helps the analysis. To meet these requirements, we have converted a series of captured code snapshots to the file format used in the CVS (Concurrent Versioning System) [1]. By importing the data into a CVS repository, various tools developed for CVS can be utilized. In particular, we used the ViewVC tool for visualizing change history and source code “diff”s between versions. We did not use any other tool for defect detection.

To familiarize ourselves with the code, we looked at both the initial and final versions for all students. Since the data came from multiple students solving the same problem, the understanding of one code helped understanding other students’ code. In some classes, students were given skeleton code to start from, which made the code understanding easier as basic code structures tended to be similar across students.

To look at changes, we examined all versions when the number of versions was relatively small. Some students compiled the code more than 600 times, in which case we picked important versions according to the strategy described in Section 4.1. Since we captured the source files every time the subjects compiled them, the number depended on their work behavior; Some students compiled code quite often, even when sometimes they did not make any changes, while other students compiled the source code far less often.

Before doing the analysis, we had prior knowledge that several types of defects were considered important in the HPC community. For example, we knew that synchronization defects such as deadlocks and races occur in multi-processor/thread programs in general, so we looked specifically for these types of defects. Also, we kept in mind that program performance is an important requirement in HPC. While producing correct output with poor performance may not be considered a defect in other domains, we had to look for problems in the code that would lead to unsatisfactory execution speed.

Figure 3 shows an example of a defective MPI code fragment, which was taken from actual data from students' code for the Buffon-Laplace Needle Problem, which is a method for approximating π using Monte Carlo simulation (the code was simplified to illustrate an important part). In this program, a pseudo-random sequence (rand()) is used to simulate random trials. To use independent sequences, the processors must initialize the sequence with different seeds in the srand() function. However, since the time() function returns the same current time in seconds, which is likely to be the same for all processors, they end up with the same pseudo-random sequence. Using the same seed for all processors is a defect that causes the loss of randomness, which reduces the accuracy of the result.

```
int np;
status = MPI_Comm_size(MPI_COMM_WORLD &np);
...

srand(time(NULL));

for (i=0; i<n; i+=np) {
   double x = rand() / (double)RAND_MAX;
   if (…) ++k;
}
status = MPI_Reduce( ... MPI_SUM... );
...
return k/(double)n;
```

Figure 3. Example of a defective MPI code.

Below are other examples of the identified defects during the initial defect analysis.

- In an MPI program, the MPI_Finalize() function must be called before the program exits. A common defect was to forget to call MPI_Finalize() in some execution path.
- When parallelizing a sequential program by dividing the problem into processors, loop boundaries often have to be modified to reflect the change in the mapping logic to the problem space from the sequential program. Incorrect modification leads to a defect, and is observed as out-of-bounds errors, slightly incorrect output, etc.
- In a program employing the "master-worker" model [12], the master process (typically the rank 0) is just waiting while other "worker" processors are executing the loop. This is a performance defect.
- Inter-dependencies between the processor communications lead to significant performance problem, as the processors are forced to follow unnecessary scheduling constraints.
- Point-to-point communication functions (MPI Send and Recv) must be coordinated to avoid deadlock. A program that depends on the assumption that the MPI_Send() never blocks causes a potential deadlock depending on the MPI library implementation and message size. [13] This type of defect is harder to detect since it doesn't always occur.

4.3 Documenting Defects

4.3.1 Methodology

The record of identified defects is the direct output of the analysis. Since our methodology is bottom-up, all the high-level knowledge to be derived from the analysis depends on how much information the low-level defect records contains Therefore, it is extremely important that the analyst record all findings regarding the defects and related contexts.

To make documentation systematic, we recommend preparing a template form which the analyst can fill in to document the findings. The appropriate format of the template form will vary with the target domain, but should contain the following information.

- Problem being solved by the program (**problem**)
- Where was the defect found: file and version (**location**)
- What was wrong in the code (**fault**)
- How the defect manifested itself (**failure**)
- When the defect was inserted into the code and when it was fixed (**time to find and fix**)
- How the defect was investigated and fixed by the developer (**developer workflow**)
- Other findings and contexts (**description**)

In addition, all known context variables should be recorded. Such variables include the language used, the nature of the problem being solved, distinctive patterns in the code change, and debugging techniques used.

A strength of this method is the richness of the information generated by the analyst. The degree of detail that can be achieved varies depending on the maturity of the defect analysis. Not all information has to be provided at once, since the collection of records can be iteratively revised as more results become available. For example, *location* and *fault* will be easier to identify initially than *failure* and *time to find and fix.*

4.3.2 Case study in HPC

In the case study, each item describe above was recorded in free text. For example, the record for one instance of the defect shown in Figure 3 is as follows.

- **Problem**: Buffon-Laplace in MPI + C. The students first wrote a serial program and parallelized with three different languages.
- **Location**: ES04-A1-05, revision 22, line 28: srand(time(0));
- **Fault**: srand() was called with the same seed (time() returns current time in seconds, whch is likely to the same across processors)
- **Failure**: the accuracy of the output is less than expected. This is hard to detect, as it is an approximation problem
- **Time to fix**: fixed in revision 23, took 6 minutes

- **Workflow**: not sure how the subject investigated the defect
- **Description**: (omitted, it is identical to the description for Figure 3)

4.4 Classifying Defects

4.4.1 Methodology

Although the individual defect samples recorded in the previous subsection are useful by themselves, they should be classified into categories in order to integrate the obtained knowledge and identify patterns, which can provide insight in various ways:

- By grouping similar defects together, we can quantitatively analyze the defect occurrence and severity by type. This allows comparison across data from different projects and different languages.
- Classification provides abstracted knowledge of defects in the target contexts
- Presenting defect classification enables feedback at a higher level of abstraction without forcing verifiers to reread the code itself

To develop a good classification scheme from the given defect set, the analyst must consider various aspects of the defect data. We recommend a bottom-up approach of grouping the defects by common context information or descriptions. Once good groupings are obtained, the analyst should develop a definition for each group by extracting common characteristics of defects and abstracting them. Defining a classification scheme will help the analyst decide whether the analysis was sufficient. If a good classification scheme cannot be defined, the analyst should go back to the defect analysis.

There can be more than one classification scheme. Different classifications may be possible from same defect dataset depending on who classifies them. Useful classification schemes depend on the specific goals, so an appropriate level of abstraction and the focus of the schemes should be determined based on the purpose of the defect analysis.

Good classification depends on the purpose of the classification. General properties include orthogonality (each defect fits into only one type), completeness (for all defects of interest, there is always a place to put it), and consistency (different analysts classify a defect into the same defect type).

Within the methodology, classification is useful as a place to get high-level feedback for the results of the analysis. Presenting classifications with concrete defect examples will help clarify the results. For correct understanding, wording is important in defining each classification scheme.

4.4.2 Case study in HPC

To classify the defects that were identified and recorded, we organized them into a candidate set of groups. Since the data we obtained came from multiple students solving the same problem, we had multiple examples of very similar defects. This made the initial grouping easier, as very similar defects were made by multiple students, which form an obvious group. For example, when we try to abstract the defect instance in Figure 3, we paid attention to the fact that this particular code portion did not contain any constructs of a parallel language. The code was indeed correct as a serial program. So this defect can be conceived as an instance of the defect type in which ordinary serial language constructs can cause a defect when they are put in a parallel context.

In addition, we have reviewed literature on debugging tools for HPC [4][10][17] that describe the defect types they assume. Although their classification cannot be directly applied to our data as they cover only the defect types their tools are targeted for, they were useful for considering possible grouping. Finally, the definitions for the abstracted defect types were created.

Table 1 shows the initial classification scheme we defined.

Table 1. Initial defect classification scheme

Type	Definition
Algorithm	Logical error
Serial constructs	Defects also seen in a sequential program
Parallel language features	Misuse or failure to use language feature
Problem space decomposition	Incorrect/improper decomposition
Synchronization	Incorrect/unnecessary synchronization
Load balancing	Unbalanced workload for processes/threads

4.5 Developing Hypotheses

As the analysis progresses, a set of research hypotheses will be generated from the analysis results. These hypotheses should capture high-level characteristics of the defects, e.g., which defect types are frequently observed, which takes more effort to fix, etc. The goal is to give support to domain-specific research questions.

- How can this defect type be detected? (detection logic)
- What can be done to avoid this defect type? (advice)
- What kind of tool is effective to prevent this defect type? (tool)

In the case study, we have not yet developed formal hypotheses as further iterations of the analysis need to be carried out.

5. VERIFICATION

5.1 Methodology

Although human analysis is very powerful, we expect a certain degree of variation in accuracy across analysts and even within analysts over time. An analyst's criteria for what is a defect may vary over time. It is also possible for an analyst to make an error. These are why it is important to verify the results of the analysis. Verification can be performed at several different abstraction levels. Each approach requires a slightly different skill set, and the cost of verification depends on the target domain and the maturity of the analysis.

5.1.1 Quantitative verification

Input: defects (identified and true sets), output: precision and recall

If we can somehow obtain the "true" defect sets, we can directly compare the analysis results with them to evaluate the analysis results quantitatively. There are two standard measures for this kind of evaluation.

- Precision: the ratio of the actual defects to the defects identified by the analysis. Defined as 1 – (false positive ratio).
- Recall: the ratio of the defects identified to the number of total defects that exist in the code. Defined as 1 – (false negative ratio).

Unfortunately, getting a “golden-truth” set of defects situation is often difficult in practice. A possible way to conduct this kind of verification is to start with a known defect set. The verifier calculates precision and recall by comparing the defects identified by the analyzer with the known defect set.

5.1.2 *Individual defect based*

5.1.2.1 *Reliability study*

Input: source code, output: defects

In this approach, multiple analysts independently analyze the source code and record identified defects. The analysts act as their own verifiers, as their results are compared against each other to measure agreement. Several measures of inter-observer agreement exist in the literature (e.g. index of agreement, Cohen’s Kappa) [15].

5.1.2.2 *Review of defects*

Input: defects + source code, output: support/deny

In this approach, verifiers examine individual instances of defects to check if each defect is correctly captured and documented. They can also provide additional insights on each defect instance. Verifiers are expected to be more familiar with the code inspection, as they may need to refer to the source code to evaluate the description of individual defects. This can be done by an interview with experts.

5.1.3 *Classification scheme based*

5.1.3.1 *Reliability study*

Input: defects and classification scheme, output: classified defects

In this approach, One or more verifiers are provided defect instances and asked to classify them into one of the given defect types. This type of approach can also be used to check the consistency of the classification scheme.

5.1.3.2 *Review of classification*

Input: classified defects, output: support/deny

In this approach, verifiers are presented with the definition of a defect classification scheme and asked to provide feedback. This is an effective way to validate the analysis results as a whole, because the verifiers can check if all important defect types are covered in the scheme without going through individual defects or the source code itself. They can point out if the classification scheme should be modified, and/or if a new defect type should be added. Again, they are expected to have generic knowledge about recurring defects from their experience.

This can be done by either an interview or a survey. Verification at this level of abstraction is often useful, as the workload of verifiers is relatively small, yet it can still provides insights on the low-level analysis results.

5.1.4 *Hypothesis based*

5.1.4.1 *Review of hypothesis*

Input: hypotheses, output: support/deny

In this approach, the verification is performed by either supporting or denying the hypotheses based on the verifiers’ experience. In other words, the verifiers act as experts who review the output of the entire defect analysis and provide feedback. This can be done by an interview or a survey.

5.1.4.2 *Experimental verification*

Input: hypotheses, output: result of a controlled experiment to test the hypotheses

To test a specific hypothesis directly, a controlled experiment can be conducted. An appropriate setting of the experiment depends on the hypotheses being tested. Some experimental results can be obtained without involving human verifiers, but experts' opinions are often useful in interpreting the results and developing more sophisticated hypotheses.

5.2 Verification in the Case Study

The verification was performed on the level of classification scheme as well as defect data review.

5.2.1 *Review of classification by survey*

To verify our results, we first conducted a survey. The participants of this survey were programmers, scientists, and vendors of HPC technologies who were attending an HPC-related meeting. For the survey, we first explained our classification scheme and defect examples in an oral presentation. Then we distributed paper survey forms. We also prepared a web version of the same survey so that the participants who preferred entering the answers online could do so. The survey was anonymous, and 14 people returned the form. Their experience in HPC development varied from 5 to 22 years.

The survey form consisted of several questions. The survey question intended to validate the defect analysis results at the classification level was as follows: “*Do the defect types [shown in Table 1] look reasonable? Can you think of other defect types, better wording or modification in definitions? Can you suggest a different classification scheme?*” The answers were given in free text form.

For the proposed defect types, most participants agreed on our classification scheme and suggested no modifications. The comments provided were as follows:

- One participant commented that in his projects, the algorithmic defect is eliminated by peer review before any code is written.
- One participant commented that he did not understand the serial construct defect. He noted that he did not listen to the presentation.

For additional defect types, the following comments were provided.

- Three participants suggested defects related to I/O. The proposed defect types include I/O data format/conversion errors, I/O performance issues, resource mismanagement in file/socket open, and generic I/O problems.
- Four participants suggested defects related to memory management. The proposed defect types include memory mismanagement, invalid memory operation errors, memory placement performance issues, and memory contention.

5.2.2 Review of defects + classification by interview

The second validation was performed at the level of individual defects as well as the classification level. We interviewed with a professor who has taught an HPC course for several years, along with his teaching assistant who has been involved in multiple iterations of the course. The professor has many years of experience in HPC development.

The interview was conducted over the phone. Before the interview, we prepared presentation material containing concrete examples of defects categorized by type. During the interview, we went through each defect example and examined if it looked reasonable. The interviewees agreed that all defect examples were observed quite often. They also agreed the classification scheme was reasonably defined, but they commented about wording as follows:

- The defect type "serial constructs" is confusing. A suggested name is "side-effect of parallelization."
- The defect type "load balancing" is too specific, as some defects categorized as this type are addressing different kind of performance problem (scheduling). A suggestion is to rename it to "performance" to accommodate both kinds of problems.

Finally, they pointed out a defect which existed in the code presented in Figure 3 but had not been identified in the previous analysis. In the defect with the use of a pseudo-random sequence, the implementation of the rand() function causes hidden serialization. It leads to a performance problem when this function is called by many processors simultaneously.

Based on the feedback from the verifiers, the classification scheme has been updated as shown in Table 2. In a new scheme, while the number of defect types was kept the same, sub-types were defined to reflect the feedback from verifiers. Potential defect types related to I/O are put as a sub-type of the "side-effect of parallelization" defect in this revision. The definitions for several defect types were rewritten for clarification. Using this scheme and all other knowledge gained by the feedback from verifiers, the source code should be reexamined in the next iteration to check if the defects that were uncaught in the previous iteration can now be identified.

Table 2. Revised defect classification scheme

Type	Sub-type	Definition
Algorithm		Logical error
Side-effect of parallelization	File I/O Random func	Serial constructs causing correctness and performance defects when accessed in parallel contexts
Use of language features		Erroneous use of parallel language features
Space decomposition		Incorrect mapping between the problem space and the program memory space
Synchronization	Deadlock Race	Incorrect/unnecessary synchronization
Performance	Load balancing Scheduling	Scalability problem because processors are not working in parallel

6. SUPPORT BY TOOLS

One obvious challenge of reading-based defect analysis is that manual reading of code is labor-intensive. It simply takes a significant amount of time when a large program is analyzed, or when there are many projects to be analyzed. Furthermore, since a larger program tends to be harder to understand, the task of finding defects by reading the source code becomes more challenging.

The use of existing tools accelerates the detection process considerably, when such tools exist. However, these tools are not available for the majority of the defects we are interested in. One solution is to develop a tool that can detect certain type of defect. The development of these tools takes effort, and therefore a tradeoff is implied. The analyst must decide where to invest in automation. The following points should be considered.

- Is the automation feasible? To develop a defect detection tool, a clear definition of the defect is required. Even if the definition is clear, automation may not be possible. If not, using knowledge as heuristics is more appropriate.
- Costs and benefits regarding the nature of the defects
  - Does the defect seem to be recurring?
  - How much effort is required to automate the detection process?
  - Should we implement the detection logic from scratch or combine existing tools depending on the context?
- Costs and benefits after the tool is developed
  - How expensive will the tool be to use? Does the code have to be executed? Does the code have to be compiled? Is special input required? How long will it take to run the tool? What language can it be applied (generalizability, applicability)?
  - What other use does the tool have? Good tools are often useful as defect detection tools for developers too.

These decisions are dependent on the contexts of the analysis. Since the analysis is iterative, tools can be built during the analysis process to gradually increase the degree of automation.

Other kinds of tools, including preprocessors and visualization tools, can improve the accuracy and efficiency of manual analysis by automating routine processes and allowing the analyst to focus on the high-level tasks of defect analysis. For example, our experience shows that a "diff" tool, which can visually display the difference in the source file between versions, is particularly useful for manual code analysis. Again, the requirements for these tools may not be known before the analysis, and they should be revealed as the iterative process goes on.

7. DISCUSSIONS

7.1 Requirements

As in any methodology, there are prerequisites to use our methodology.

7.1.1 Availability of source code

The first requirement is that there is source code available for analysis. Since many projects use a code management system such as CVS (Concurrent Versioning System), source code is usually available. In most cases, what are available are the "debugged" versions of the code. We are interested in the intermediate versions that exist between debugged versions. However, if we cannot have all versions, we can look at the differences from one debugged version to the next.

Also note that there are different degrees of availability for source code. The most complete data is a complete set of source files with all the change histories. The minimal case is a partial source file of one particular version. The degree of availability affects the efficiency and completeness of the analysis.

7.1.2 Availability of analysts
Since our methodology is based on code reading, our second requirement is the existence of one or more personnel who can actually inspect the source code, identify defects and record them. Locating qualified analysts may be difficult. For one thing, it is a well-known phenomenon that programmers dislike reading code written by others, which may explain why inspection methods are frequently not used despite the evidence that they are effective. It is also a time-consuming job, so it is necessary to evaluate the cost of "hiring" analysts. The skill of the analysts is also very important. Analysts must have sufficient knowledge and experience to be able to read other's code and find problems. This requires deep understanding of both the language and the problem being solved. The analysts must familiarize themselves with the structure of the code being analyzed quickly enough.

7.1.3 Availability of verifiers
The third requirement is the existence of verifiers who can examine the analysis results, provide feedback, and validate results as described below. The existence of verifiers is indispensable to make this methodology repeatable and verifiable. However, they may be gathered during or after the defect analysis.

The qualities required in verifiers are similar to those for analysts. However, if the verification is conducted at a higher level, verifiers may not have to directly inspect the code itself. Instead, they need to have tacit knowledge about recurring defects to provide proper feedback.

7.2 Lessons Learned
The results of the case studies provided the following insights.

- The study indicates the basic feasibility of the reading-based analysis in domain-specific contexts. It was successfully applied to identify several defects recurring in students' code and derive the classification scheme.
- We found that the use of source file diff tool is vital for an efficient manual analysis. For example, by comparing an intermediate version with the final version, we can identify the changes that should contain all the bug fixes between them. This helps determine what portion of the code should be examined when there is no other clue available.
- Both the survey and the interview successfully provided information useful for verifying the defect classification scheme. In general, the verifiers agreed on majority of the analysis results. Their feedback helped the analyzer revise the classification scheme and improve the subsequent defect analysis. The classification scheme helped reveal a defect type that had not been addressed by the analysis.
- The verifiers understood the classification scheme better when they were also presented concrete defect examples for each defect type. Providing only the definitions of the defect types without examples confused some verifiers and made the classification scheme harder to understand.
- The defect analysis was labor-intensive especially when there were no heuristics or tools available. Analysts can easily get lost without a specific perspective to look at the code. As the analysis progresses, the analysis can be made more systematic. This seems to support the assumption that the analysis is difficult at the startup, and analyzing simpler code first is recommended.
- Aside from the fact that change history is essential for determining how much time was spent to find and fix each defect, looking at the change history was useful for just identifying defects. For example, by comparing an intermediate version with the final version, the analyst can determine what has been fixed between two versions. This technique is especially useful when the analyst is trying to determine what aspect of the source code should be examined carefully. In practice, a good visualization tool that can graphically display the difference between versions improved the efficiency of the manual analysis.
- Understanding of the problem domain seemed to have strong influence on the efficiency of the analysis. This may cause additional workload to the analysts. In HPC, it is common that understanding of the core algorithm requires the background knowledge on the underlining science and mathematics. When an analyst needs to examine the code from multiple problem domains, it is probably necessary to get help from project developers who are already familiar with the code and the algorithm.

8. SUMMARY
In this paper, we described a methodology of defect analysis based on code reading. Our methodology uses existing software with source code history, and it can be used to identify defects and construct a defect classification scheme by analyzing inspecting source code history. The methodology depends upon the availability of source code for defect reading analysis. Many organizations use a source code management system, so change history is more likely to be available than bug history. It also depends upon developers involved in the software having tacit knowledge about recurring defects from their own experience. Even if it is not easy for them to package their knowledge in a usable form, if they are presented concrete material prepared by someone else, they can respond to it by saying whether they agree or not, and modify the material with new perspectives. This is most successful when they are shown actual examples related to their own experience.

We have applied our methodology to the HPC domain, where we need to build knowledge on recurring defects to decrease development cost. We collected data from graduate students who were learning HPC programming and solved some HPC problems as class assignment. The collected defect data should provide baseline data for novices and small problems, which will be useful to move forward to larger, more complex problems with more experienced developers. We have analyzed the code snapshots from 38 students in 5 problems solved with MPI-C, and identified defects. From this data we have developed a defect classification scheme.

For validation, we have done classification-level validation and validation at the defect level. At the classification level, we have gotten responses which indicate that our defect classification scheme was reasonably defined. However, people have different opinions about which defects are important, possibly because their projects have different contexts. Several comments provided lead possible refinements to the classification scheme in the future as we analyze more data. At the defect level, the defects we have identified in HPC code were confirmed to be recurring. Furthermore, the verifiers pointed out another defect in the example that was not previously identified by the analyst. All of the information obtained at both abstraction levels obtained as a response to the analysis results was very useful, and thus, our methodology indeed worked well for turning hidden knowledge of experts into explicit knowledge.

Although our study is preliminary, the results indicate the feasibility and effectiveness of the methodology. In a future, we will do more case studies to obtain more results so that we can refine the methodology.

ACKNOWLEDGMENTS

This research was supported in part by Department of Energy contracts DE-FG02-04ER25633 and Air Force grant FA8750-05-1-0100 to the University of Maryland. We wish to acknowledge the contributions of the various faculty members and their students who have participated in the various experiments we have run over the past 2 years. This includes Alan Edelman at MIT, John Gilbert at the University of California Santa Barbara, Mary Hall at the University of Southern California, Alan Snavely at the University of California San Diego, and Uzi Vishkin at the University of Maryland. We wish to thank Stacy Hochstein for reviewing the draft of this paper.

REFERENCES

[1] Bar, M. and Fogel, K., Open Source Development with CVS. Paraglyph, 2003.

[2] Basili, V. R. and Perricone, B. Software Errors and Complexity: An Empirical Investigation. Communications of the ACM, Volume 27, Issue 1, 1984, 42-52.

[3] Basili, V. R. and Reiter, R. W. Jr. Evaluating Automatable Measures of Software Development. In Proceedings on Workshop on Quantitative Software Models. October, 1979.

[4] Collette, M., Corey, B., and Johnson, J. High Performance Tools & Technologies, Technical Report UCRL-TR-209289, Lawrence Livermore National Laboratory, US. Dept. of Energy, 2004.

[5] Dagum, L. and Menon, R. OpenMP: An Industry-Standard API for Shared-Memory Programming, IEEE Computational Science & Engineering, Volume 5, Issue 1, 1998, 46-55.

[6] Dongarra, J. J., Otto, S. W., Snir, M., and Walker, D. A Message Passing Standard for MPP and Workstations, Communications of the ACM Volume 39, Issue 7, 1996, 84-90.

[7] Dunsmore, A., Roper, M., Wood, M. Practical Code Inspection Techniques for Object-Oriented Systems: An Experimental Comparison, IEEE Software, Volume 20, No. 4, 2003, 21-29.

[8] Dunsmore, H. E. and Gannon, J. D. Programming Factors - Language Features That Help Explain Programming Complexity. In *Proceedings of the 1978 Annual Conference, ACM/CSC-ER*, ACM Press, New York, NY, 1978, 554-560.

[9] Hovemeyer, D. and Pugh, W. Finding Bugs Is Easy. ACM SIGPLAN Notices, Volume 39, Issue 12, 2004, 92-106.

[10] Luecke, G., Zou, Y., Coyle, J., Hoekstra, J., Kraeva, M. Deadlock Detection In MPI Programs, Concurrency and Computation: Practice and Experience. Volume 14, 2002, 911-932.

[11] Maldonado, J., Carver, J., Shull, F., Fabbri, S., Doria, E., Martimiano, L., Mendonça, M., and Basili, V. Perspective-Based Reading: A Replicated Experiment Focused on Individual Reviewer Effectiveness. Empirical Software Engineering: An International Journal. Volume 11, Number 1, 2006.

[12] Mattson, T.G., Sanders, B.A., and Massingill, B.L. *Patterns for Parallel Programming*, Addison-Wesley, 2004.

[13] Message Passing Interface Forum, MPI: A Message Passing Interface Standard, May UT-CS-94-230, 1994.

[14] Numrich, R. W. and Reid, J. K. Co-Array Fortran for Parallel Programming, RAL-TR-1998-060, 1998.

[15] Robson, C. *Real World Research*, Blackwell Publishing, 2002.

[16] Spacco, J., Strecker, J., Hovemeyer, D., and Pugh, W. Software Repository Mining with Marmoset: An Automated Programming Project Snapshot and Testing System, In Proceedings of the Mining Software Repositories Workshop (MSR 2005), St. Louis, Missouri, 2005

[17] Squyres, J., DeSouza, J. Why MPI Makes You Scream! And how Can We Simplify Parallel Debugging? BOF in Supercomputing'05 (SC05), 2005.

[18] Top 500 Supercomputer Sites, http://www.top500.org/

[19] ViewVC (ViewCVS) Repository Browsing, http://www.viewvc.org/

[20] Weiss, D. and Basili, V. R. Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory, IEEE Transactions on Software Engineering, Volume 11, Issue 2, 1985, 157-168.

[21] Williams, C. C. and Hollingsworth, J. K. Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques. IEEE Transactions on Software Engineering, Volume 31, Issue 6, 2005, 466-480.