Model-Based Debugging

- Position Paper -

Salman Mirghasemi and Claude Petitpierre

School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne, Switzerland
{salman.mirghasemi,claude.petitpierre}@epfl.ch

Abstract. Software Debugging is still one of the most challenging and time consuming aspects of software development. Monitoring the software behavior and finding the causes of this behavior are located at the center of debugging process. Although many tools and techniques have been introduced to support developers in this part, we still have a long way from the ideal point. In this paper, we first give a detailed explanation of the main issues in this domain and why the available techniques and tools have been incapable of solving these issues completely. Then we explain how employing models can be helpful in solving stated problems. Finally, a detailed sketch of our approach based on using runtime models of executing software is described.

1 Introduction

Software Debugging has little changed during the past decades and has remained one of the most challenging and time consuming aspects of software engineering [1]. Debugging is much more an art and the predominant techniques for finding bugs are still data gathering (e.g., print statements) and hand simulation. Software developers spend huge amounts of time, up to half of their time, debugging. Fixing and finding bugs faster and more effectively directly increases productivity and can improve program quality by eliminating more defects with available resources [2].

In a larger view, software debugging has been usually considered as part of software testing and it has not been studied enough until recent years. Research is quite young in this area and most studies has been done in the last decade. Table 2 shows a big picture of different stages of software verification, testing and debugging. Having this picture can help us to better understand the close relation of debugging with testing and verification and how it differs from them. All these stages deal with the behavior of the software and recognizing the appropriate behavior from inappropriate one. But they are different in goals. As it is written in the first row of the table, the main goal of software testing and verification is finding software bugs or equivalently, answering to this question:" *Is there any bug in the software?*". Software verification tries to prove software correctness but software testing tries to make program fails, different approaches

	Software Verification, Testing and Debugging					
Goal	Answring to this question: Is There Any Bug?		Fixing Found Bugs			
	Proving Correctness	Finding Bugs	Reproducing Bugs	Simplifying/Isolating Bugs	Finding Defects	Fixing Defects
Approach	Formal Verification	Testing	Reproducing the Envirnoment Reproducing the Execution	Removing All Irrelevant Circumstances	Scientific Debugging Algorithm	
Technique	Model Checking Theorem Proving	Black Box Testing White Box Testing		Delta Debugging	Monitoring Software Behaviour Finding Causes	
Do Models Help?	Yes	Yes	?	?	?	?

Table 1. Using Models in Different Stages of Software Verification, Testing and Debugging

Issue	**Description / Problems**	**Related Work**
Interactive Execution	The ability to go forward and backward in an execution.	Capture and Replay
	Re-execution is time-consuming and painful, Every execution might change the program state, Nondeterminism	Controlled Execution
Specifying Interesting Execution Events	Before every execution a developer has to specify events that he intends to observe.	Printing Statements Breakpoints Aspect Pointcuts
	Available languages are not enough expressive, Requires compilation and cannot be changed in runtime, A developer has to exactly declare which data should be collected at the specified moment (excepting breakpoints).	
Querying Program Execution	To find the possible origings of an incorrect result, a developer has to search in space and time.	Alpha JIVE
	Limited support only at code level, no support at runtime.	
Easy to Use Interface and Scalable Data Visualization	A debugging user interface which provides scalable data visualization and ables the developer to easily navigate through different views can greately help the developer in observing facts.	Omniscient Debugger WhyLine JIVE
Automation	Automatically recognizing wheather a result is correct or incorrect, Automatically finding a list of possible causes of a behavior	WhyLine

Table 2. Main Issues in Finding Defects

for answering the same question. On the other hand, the goal of debugging is fixing the found bugs.

Debugging process can be separated to four stages. *Reproducing the bug* is the first step in debugging.It is difficult to find and fix a software problem, and to verify the solution, without the ability to reproduce it [3]. The next step is *simplifying the bug* that each part of test case be relevant. This step is not necessary but it can be very helpful. *Delta-Debugging* [4] is a technique introduced for bug simplification. The main part of debugging is finding the defects. As is well known among software engineers, most of the effort in debugging involves locating the defects. Finally, the located defects should be fixed.

Employing models have always been useful in different stages of software development including software testing. In recent years the greatest attention in software testing has been turned to *model-based testing* [5]. We believe that similar to other areas, debugging can also benefit from using models. In this paper, we explore this idea and propose *runtime models* for describing an executing program.

2 Problem Description

Once a developer knows how to reproduce a bug, he enters to the locating defects phase. To better figure out the needs of a developer in this phase, we have to reconsider the locating defects process. This process is a heuristic process, in which a developer goes through a chain of act-observe steps and its goal is locating the defect.

To model debugging process, we use *scientific debugging* algorithm introduced by Zeller [2]. Based on this algorithm, at every step, a developer has defined a set of possible origins for an incorrect result. For every origin, she has to specify whether it is correct or incorrect. If an origin is incorrect, the developer has to repeat the steps with this origin as the incorrect result. This algorithm continues until the developer locates the main defects. Although not all developers know this algorithm, they implicitly go trough the same process for debugging.

According to this description, a developer has two kinds of problems in the debugging process. *Specifying a set of possible origins for an incorrect result* and *specifying whether an origin is correct or incorrect.* But how developers solve these two problems? The main approach is the *observation of program execution.* This is what makes *reproducing bug* a necessary step before debugging.

The first issue in locating defects arises here. For every step in the algorithm, a developer needs to observe the program execution and therefore many observations are required for finding a defect. For every observation the program should be executed another time. But re-executing the program can be time-consuming and painful, because an execution might take a long time and require many actions from the developer. In addition, there are many cases that every program execution changes the program's data and the developer has to rollback the data to the initial state, which is not usually an easy task. Moreover, because of many non-deterministic conditions during the execution, the new execution might go through another path and therefore developer observes a different execution.

To understand other issues we have to consider the observation activity. The observation of program execution consists of observing software artifacts, runtime and collected data of execution. Developers have to usually specify interesting moments and events during an execution before the execution. For example a developer defines breakpoints in a debugger or put printing statements in parts of code to specify the interesting points for observation. Most of the current methods require compilation and re-execution of software. In addition the current methods aren't enough powerful to let a developer to exactly define the interesting moments. The second issue is specifying interesting parts of execution and the data should be collected at these parts.

Once a developer sees an inappropriate behavior, she must then translate her questions about the behavior into a series of queries about the programs code. She has to search in space and time which becomes the more difficult the larger the program state, and the distance between defect and failure. The third issue is the need for querying the runtime state and the history of execution. During the program execution, huge amount of data might be collected and analyzing this amount of data and understanding the relation of each record to time and space by a developer requires a well-designed user interface which supports scalable data visualization. Finally, the last issue is debugging process automation and how parts or the all steps of debugging process can be automated?

3 Related work

In this section, we explore related work in this area and how each one of the mentioned issues are addressed in other works.

3.1 Interactive Execution

There are two approaches for attacking this problem. The first one is the *controlled execution* of program. It means that we try to drive the program execution through the same path executed before. This is mainly done by eliminating the non-deterministic conditions (e.g., thread interleavings) during the execution. This idea is used in *CHESS* [6] for finding and reproducing *Heisenbugs* in concurrent programs.

The next approach is named *Capture and Replay.* In this approach, numerous events will be stored during the execution and later, these events can be used for replaying the execution. For example *Omniscient Debugger* [7] works by recording all state changes in the run of a program, and then allowing the programmer to explore the history of that program going backwards in time.

Capturing and replaying interaction is not only a helpful tool for reproducing failures, it can also be helpful for isolating those events that were relevant for the failure by combining capture/replay with isolation of relevant events. JINSI [8] is a tool which uses this approach. Unfortunately, capturing complete executions is generally infeasible, for several reasons. First, there are practicality issues. To capture a complete execution, we may need to record a large volume of data and all the inputs to the application. Also, capturing the inputs provided to an application can be difficult and may require custom mechanisms, depending on the way the application interacts with its environment. Second, the environment may have changed between capture and replay time [9].

3.2 Specifying Interesting Execution Events

Using print statements and breakpoints are two primary methods used by developers to specify a moment in the execution. Each one has its own characteristics. In case of using print statements, the developer has to specify the data should be written but in breakpoints this step is not required and the developer has access to all data in memory. When a developer uses breakpoints he has to drive the program execution herself and this is a tedious task for long time executions, but in the other case she just define print statements. Using print statements, a developer can observe all the history of execution in log files which is not possible in the other case. Both methods requires compilation after any change in the definition of interesting events. Finally when the huge amount of print statements are executed understanding the log files is not an easy task.

The joinpoint concept in aspect oriented is very close to specifying event in the execution. Aspect oriented is mainly built with idea of changing the behavior of the system and therefore the pointcut concept is defined based on the structure of a program instead of its execution state. Alpha [10] is introduced as a new tool with an extended pointcut language in the form of logic queries over different models of the program semantics. Using this language pointcuts can be defined regarding to program runtime state that can facilitates specifying interesting events during execution.

4 Ideal Debugging Environment

Figure 1 illustrates the characteristics of an ideal debugging environment. It is similar to a media player which lets the user goes forward and backward and observe different aspects of a captured execution. Also it supports querying on the execution events. For example "*show me the last moment object X has bean changed?*" or "*find methods which have returened null value.*". In addition, it should provide a user-friendly interface that let user to conveniently find and view her interesting events among huge amount of collected data.

5 Approach

One of the vital features for any developing tool or technique is the adaptability of its concepts to a developer's mental concepts. Debugging tools and techniques are not exceptions to this rule. In this regard, we belive that providing a higher

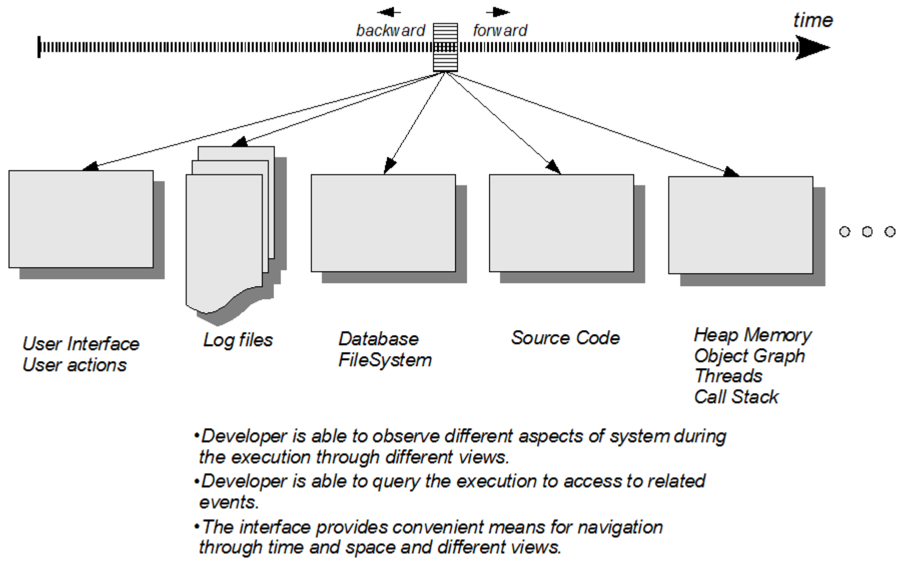
backward
forward
time
User Interface
User actions
Log files
Database
FileSystem
Source Code
Heap Memory
Object Graph
Threads
Call Stack
•Developer is able to observe different aspects of system during the execution through different views.
•Developer is able to query the execution to access to related events.
•The interface provides convenient means for navigation through time and space and different views.

Fig. 1. Ideal Debugging Environment.

level view of program execution has a significant impact on the usability of a debugging method. To support this fact, we propose using runtime models for presenting the state of an executing program.

The basic model of an execution is a finite state machine which every node is corresponding to a specified point of the execution by the developer. When a program is executing, the program will be paused at every node and a method written by the developer will be executed. This method can access any object in memory and also checks the conditions and finally returns a pass or fail result. In fact, developer will incrementally build an assistant observer which helps her to avoid redoing checks.

To more clarify the idea, we explore an example. Suppose that there is a bug in an email client. Once a user opens her address book (which has more than 50 contacts) and removes contacts one by one, the last item can not be removed. The following code shows the assistant debugger for this bug. Figure 2 shows the interface a developer gets when she debugs. Our approach attacks three issues among five mentioned issues in problem description, specifying interesting execution events by employing a more expressive language which considers runtime program state, user interface and scalable data visualization by providing a higher level view of execution which can be integrated to other views and automation by reducing a developer's repeating checks. Moreover, after fixing any bug, an artifact is available for the future, either if the same bug is reported again or another bug with similar execution context is found.

```
Debugger Bug#Num{
```

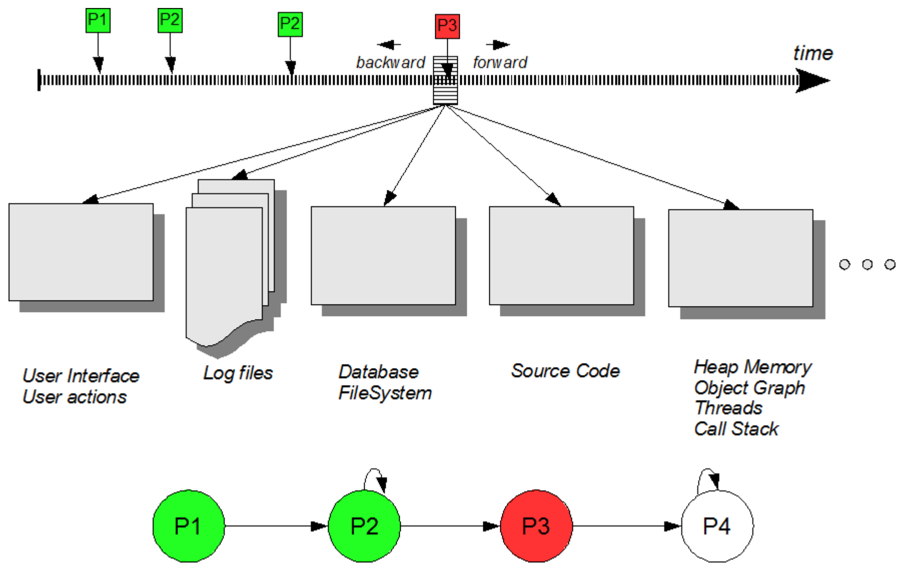

Fig. 2. The Execution State Machine

```
  // new ExecutionPoint(description, previous points, definition)
ExecutionPoint p1 = new ExecutionPoint("Window is opened" , null,
            "Window.display() and Window.tilte = 'Address Book'");
ExecutionPoint p2 = new ExecutionPoint("More than two rows" , p1|p2,
            "Table.removeRow() and Table.parentContainer.title="+
            "'Address Book' and Table.numberOfRows > 2");
ExecutionPoint p3 = new ExecutionPoint("Two rows" , p2, "Table.removeRow()"+
            " and Table.parentContainer.title= 'Address Book'"+
            " and Table.numberOfRows = 2");
ExecutionPoint p4 = new ExecutionPoint("One row" , p3|p4, "Table.removeRow()"+
            " and Table.parentContainer.title= 'Address Book'"+
            " and Table.numberOfRows = 1");

boolean p1Check(PointContext pc){
   ...
}
boolean p2Check(PointContext pc){
   ...
}
boolean p3Check(PointContext pc){
   // Monitors all changes to this object until the next point.
   Util.monitorAllChanges((Table)pc.getCurrentObject()
                                  .getModel());
```

```
        int backEndListSize = ((Table)pc.getCurrentObject())
                                          .getModel().size();
        if (size != 2)
         return false;
     }
     boolean p4Check(PointContext pc){
        if (Util.isKindOf (pc.previousPoint, p3)
        {
          ...
        }else
        {
          ...
        }
     }
}
```

References

1. Ko, A.J., Myers, B.A.: Debugging reinvented: asking and answering why and why not questions about program behavior. In: ICSE '08: Proceedings of the 30th international conference on Software engineering, New York, NY, USA, ACM (2008) 301–310
2. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann (10 2005)
3. Artzi, S., Kim, S., Ernst, M.: Recrash: Making software failures reproducible by preserving object states. (2008) 542–565
4. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. **28**(2) (Feb. 2002) 183–200
5. Bertolino, A.: Software testing research: Achievements, challenges, dreams. In: Proc. Future of Software Engineering FOSE '07. (2007) 85–103
6. Madanlal Musuvathi, S.Q., Thomas Ball, Microsoft Research; Gerard Basler, E.Z.P.A.N.U.o.W.M.I.N.U.o.C.R.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI 2008. (2008)
7. Lewis, B., Ducasse, M.: Using events to debug java programs backwards in time. In: OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM (2003) 96–97
8. Burger, M., Zeller, A.: Replaying and isolating failing multi-object interactions. In: WODA '08: Proceedings of the 2008 international workshop on dynamic analysis, New York, NY, USA, ACM (2008) 71–77
9. Orso, A., Kennedy, B.: Selective capture and replay of program executions. In: WODA '05: Proceedings of the third international workshop on Dynamic analysis, New York, NY, USA, ACM (2005) 1–7
10. Ostermann, K., Mezini, M., Bockisch, C.: Expressive pointcuts for increased modularity. In: ECOOP. (2005) 214–240
11. Utting, M.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers (2007)