Event-Based Relative Debugging

KOÇ Uğur ugurkoc@sabanciuniv.edu
DEMİRÖZ Gülşen gulsend@sabanciuniv.edu
YILMAZ Cemal cyilmaz@sabanciuniv.edu
Faculty of Engineering and Natural Sciences, Sabanci University, Tuzla – Istanbul

Abstract

Identifying the root causes of failures is one of the most time-consuming and tedious components of program debugging. Many automated approaches have been proposed to facilitate program debugging. Relative debugging is one of them. In relative debugging, a faulty program is debugged by comparing it to a reference implementation, which is assumed to be correct. Both programs are executed in parallel. The executions are interrupted at certain points during executions. The states of the programs at the point of interruption are then compared and the differences (if any) are visualized as a debugging aid. One downside of existing relative debugging approaches is that they only compare program states, without taking the history of events causing those states into account. In this work we present an event-based relative debugging approach. In this approach, we infer finite state machine models from sequences of events occurring in executions, compute the structural differences between these models, and report them as a debugging aid. A case study conducted at a small-scale shows promise.

Keywords: relative debugging, debugging, fault localization, software quality assurance.

1. Introduction

Program debugging is the process of identifying and fixing defects in programs. The most expensive component of program debugging is the identification of the root causes for failures. To this end, developers observe the symptoms of failures, develop failure hypotheses, and iteratively validate and refine their hypotheses until the root causes are located. Clearly, this process can be quite tedious and time-consuming.

Many automated approaches have been proposed in the literature to facilitate program debugging [3, 5, 7, 8, 9, 10]. The ultimate goal of these approaches is to reduce and/or prioritize the space of likely root causes for failures, which can in turn improve the turnaround time for bug fixes.

One such automated approach, which is also the focus of this paper, is *relative debugging* [1, 2]. In relative debugging, a faulty program is debugged by using a reference implementation. Both the faulty program and the reference program are executed in parallel with the same concrete input. When the control reaches to a pre-determined location in the source code, the executions are interrupted and the state of the faulty program is compared to that of reference program. The differences between the states (if any) are then visualized as a debugging aid.

Relative debugging requires a reference implementation for the program being debugged. Such reference implementations are available in many scenarios. For example, when a stable program is being ported to another platform, the stable version can be used as the reference implementation. When a parallel version of an existing sequential program is being developed, the sequential version can be considered to be the reference program. When a program is being refactored, the original program can serve as the reference implementation. The results of many empirical studies suggest that relative debugging can help developers pinpoint the root causes of failures [1, 2].

One downside of the existing relative debugging approaches, though, is that they only leverage program states for comparisons. The histories of events that have brought the programs to those states are ignored. While examining program states helps determine faulty states, the chain of events causing the faulty states needs to be traced back in order to locate the root causes. In this work we conjecture that taking program events into account when comparing executions, can further improve the effectiveness of relative debugging approaches.

2. Related Work

The idea of relative debugging was first materialized in the GUARD tool [1]. GUARD operates by comparing key data structures between a faulty program and its reference program. It takes as input a set of assertions. Each assertion determines the key data structures to be used in the comparisons and the source code locations at which the comparisons should be made. Given the assertions, program executions are interrupted at the specified locations and the differences between the key data structures (if any) are visualized. Relative debugging was later adapted for debugging of parallel programs [2].

3. Proposed Approach

In this work, rather than seeing a program execution as a sequence of program states as is the case in existing relative debugging approaches, we see it as a sequence of events. An event simply represents a high level functionality performed by the system, such as receiving/sending a network package, connecting to a server, updating a database, computing a value, spawning a process, and starting a new thread.

Events are specified by the developers of the software under test. This can be done by annotating the source code and/or by making calls to special purpose functions. Furthermore, events can be defined in a hierarchical way, i.e., an event can be composed of other events.

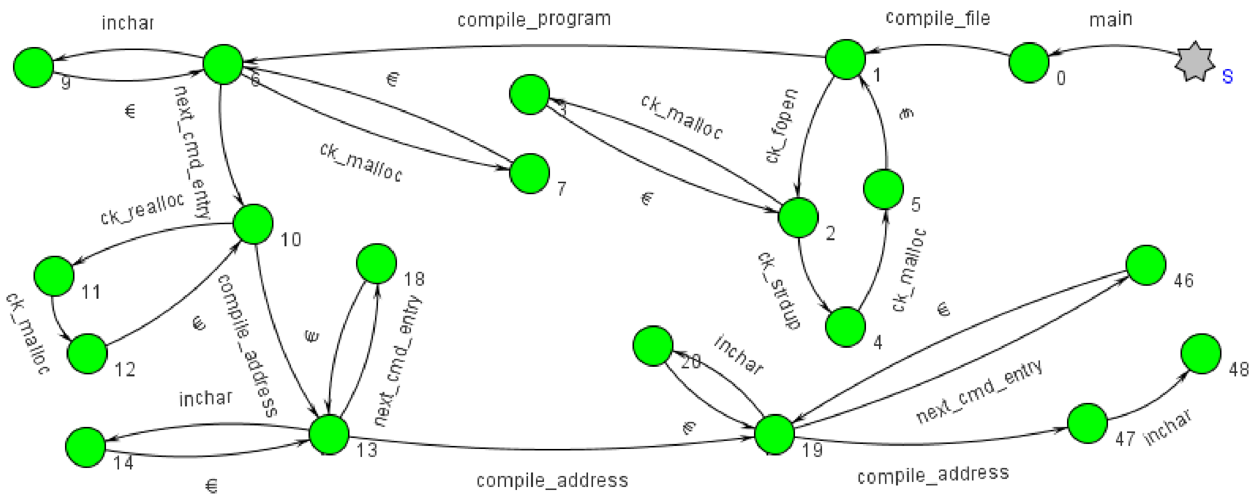

Figure 1. The FSM model inferred from the reference version of *sed*.

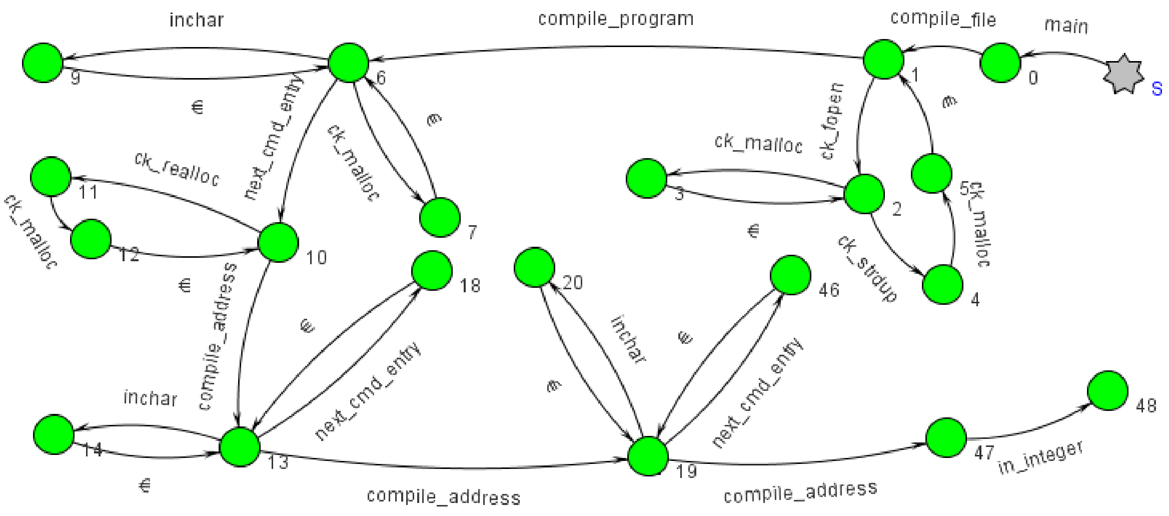

Figure 2. The FSM model inferred from the faulty version of *sed*.

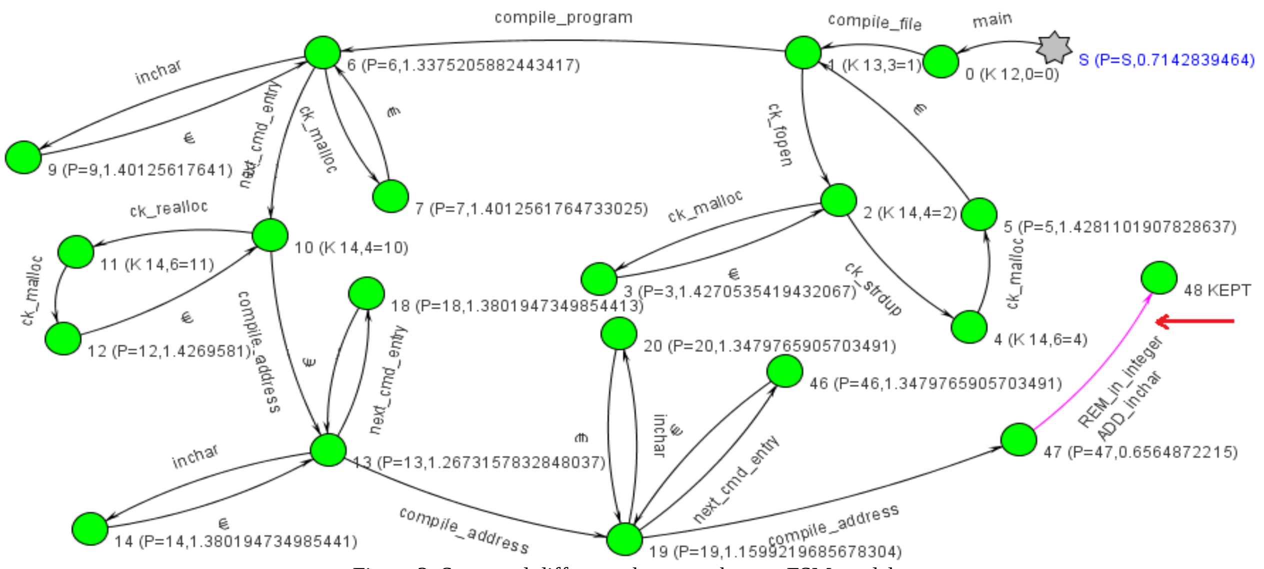

Figure 3. Structural difference between the two FSM models.

Given the definitions of events, we monitor a program execution, record the sequence of events occurring in the execution, and infer a finite state machine (FSM) model for the sequence. In these FSM models, states represent program states and transitions represent the occurrences of events. An event moves the program from one state to another.

The proposed approach can be summarized as follows. We, by following a similar approach with the existing relative debuggers, execute both the reference program and the faulty program in parallel. The executions are interrupted at pre-determined locations. For each program, we then infer a FSM model for the sequence of events that have occurred so far. Finally, we compute the *structural* difference between these two FSM models [4] and report the difference as a debugging aid.

4. A Case Study

To evaluate the proposed approach, we conducted a feasibility study. In this study, we used *sed*, which is an open source stream editor, as our subject application. The subject application, being taken from a widely-used defect repository, came with two versions: a stable version and a faulty version. We used the stable version as our reference version to debug the faulty version.

To carry out the study, we considered each function invocation to be an event. Note that, although we opted to use function invocations as events in the study, the proposed approach has a more general notion of events. We executed both programs in parallel and right before each function invocation, i.e., before each occurrence of an event, we interrupted the executions. We then inferred a FSM model per program by using the sequence of function invocations observed so far. For that purpose, we used a tool, called FsmUnitApi [6]. We then computed the structural differences between the models by using a tool, called *statechum* [4].

Figure 1 and Figure 2 visualize the FSM models obtained from the reference and the faulty version of the program, respectively. These are the FSM models computed at the time when the first difference was observed. Furthermore, Figure 3 presents the structural difference between these FSM models.

Figure 3 indicates that there had been no difference between the FSM models until the last function invocation, i.e., until state 47. However, in state 47, while the reference version was calling function *inchar*, the faulty version called *in_integer* (marked by the arrow in the figure).

The result of the analysis greatly helped us pinpoint the root cause. It turned out that the faulty program was trying to read an integer value from an input stream at a point where a character value was expected. This caused an error, which later forced the program to terminate abruptly.

5. Concluding Remarks and Future Work

In this work we presented an event-based relative debugging approach and evaluated it by conducting a case study. The result of our case study, although conducted at a small scale, demonstrates that the proposed approach is promising. However, much work remains to be done. *How should the events be specified? Could there be program debugging-oriented ways of inferring the FSM models and computing the structural differences? What types defects are best suitable for the proposed approach? Will the approach scale to larger and more complex software systems?*

In particular, we are interested in the applications of the proposed approach in incremental development scenarios. For example, consider a scenario in which, after making a change in a stable code base, some test cases, which used to work successfully in the previous version, now fail. In such scenarios, where the difference between the reference program and the faulty program is small, we hypothesize that event-based relative debugging can greatly help developers locate the root causes of failures.

References

[1] D. Abramson, I. Foster, J. Michalakes, R. Sosic, "*Relative Debugging: A new paradigm for debugging scientific applications*", The Communications of the Association for Computing Machinery (CACM), 39(11): 67 - 77, Nov 1996.

[2] D. Abramson, G. Watson, "*Relative Debugging for Parallel Systems*", In Proceedings of PCW 97, Sept. 1997, Australia.

[3] H. Agrawal, J. R. Horgan, "*Dynamic program slicing*", In Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI), 25(6):246–256, White Plains, New York, June 1990.

[4] K. Bogdanov, N. Walkinshaw, "*Computing the Structural Difference between State-Based Models*", In 16th IEEE Working Conference on Reverse Engineering (WCRE), 2009.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, D. Notkin, "*Dynamically discovering likely program invariants to support program evolution*", IEEE Transactions on Software Engineering, 27(2):1–25, Feb. 2001.

[6] A. Gargantini, M. Guarnieri, E. Magri, "*An Eclipse-based environment for conformance testing by FSMs*", 2011. Fsmunit, https://svn.origo.ethz.ch/fsmunit

[7] B. Liblit, A. Aiken, A. X. Zheng, M. I. Jordan, "*Bug isolation via remote program sampling*", In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI-03), 38(5):141–154. ACM ress, 2003.

[8] M. Renieris, S. Reiss, "*Fault localization with nearest neighbor queries*", In Proceedings of the International Conference on Automated Software Engineering, pages 30-39, Montreal, Quebec, October 2003.

[9] Y. Xie, D. Engler, "*Using redundancies to find errors*". In FSE, pages 51–60, Nov. 2002.

[10] A. Zeller, R. Hildebrandt, "*Simplifying and isolating failure-inducing input*", IEEE Transactions on Software Engineering, 28(2):183–200, Feb. 2002.