Proceedings of PhD Workshop of ICTSS’13
(co-located with the The 25th IFIP International Conference on Testing Software and Systems)

Edited by Rui Abreu

Preface

The 25th IFIP International Conference on Testing Software and Systems (ICTSS’13), held in Istanbul, Turkey on November 13 -- 15, 2013. It is as well-established ICTSS series of international conferences addresses the conceptual, theoretical, and practical challenges of testing software systems, including communication protocols, services, distributed platforms, middleware, embedded systems, and security infrastructures. Moreover, ICTSS is a forum for researchers, developers, testers, and users from industry to review, discuss, and learn about new approaches, concepts, theories, methodologies, tools, and experiences in the field of testing of software and systems.

The ICTSS Doctoral Workshop provides a forum for PhD students to present preliminary results and their thesis work and receive constructive feedback from experts in the field as well as from peers. Also it is an opportunity for researchers to get an overview of the latest research topics in the field. We invite applications from PhD students at any stage of their doctoral studies.

We received 4 paper submissions this year, out of which 2 have been selected for presentation at the workshop:

- “Runtime Verification Driven Debugging of Replayed Errors” by Hanno Eichelberger, Thomas Kropf, Thomas Greiner and Wolfgang Rosenstiel,
- "Debugging as a Service for Localizing Concurrency Faults" by Feyzullah Koca, Hasan Sozer, Rui Abreu and İsmail Ari

All papers received 3 reviews and were further assessed by the PhD Workshop Chair. We would like to thank the valuable efforts of the reviewers: Fernando Brito e Abreu, João Pascoal Faria, Natalia Kushik, Hasan Sozer, Hans-Gerhard Gross, Ana Paiva, and Arjan van Gemund.

(Rui Abreu, PhD Workshop Chair)
Runtime Verification Driven Debugging of Replayed Errors

Hanno Eichelberger1,2*, Thomas Kropf1, Thomas Greiner2, and Wolfgang Rosenstiel1

1 University of Tübingen, Computer Engineering Department, Germany, {eichelberger,kropf,rosenstiel}@informatik.uni-tuebingen.de
2 Hochschule Pforzheim University, Germany, thomas.greiner@hs-pforzheim.de

Abstract. Bugs in embedded software often depend on inputs of the target environment which are often difficult to generate for testing. Our approach traces external inputs to the software during operation until a failure occurs. The execution trace is replayed in the laboratory. During replay automated verification techniques are applied in order to check whether the software violates the requirements of the specification. A detected violation can give the developer an indication pointing to the cause of the failure in the source code. The developer does not have to step with the debugger over the whole replayed execution in order to find the reason for the failure. Thus, the analysis process is accelerated.

Keywords: Runtime Verification, Replay Debugging, In-Vehicle Infotainment

1 Introduction

The execution of embedded software often depends on sequences of user input, sensor data or the interaction with other external systems (e.g., GPS or control systems). These inputs are difficult to simulate, because they are triggered by physical world aspects or external systems from other manufacturers. Failures of the software which depend on such inputs are often not found before the first operation tests take place. When failures are detected during these tests it is difficult to locate the causing faulty operations in the source code. For being able to debug them, the original execution sequence must be reconstructed in the laboratory. This reconstruction can often only be achieved by time-consuming trial and error.

For overcoming the mentioned issues, our approach traces the input to embedded software during operation. Thus, failure-causing input sequences can be replayed in the laboratory without trial and error. During the replay of the execution automated verification techniques are applied in order to locate the cause of the failure in the source code. In this way, the analysis process is accelerated.

* This work is funded by the State of Baden Wuerttemberg, Germany, Ministry of Science, Research and Arts within the scope of the Cooperative Research Training Group
2 Related Work

Our approach combines fundamental concepts of the separate approaches Replay Debugging (RD) [1] and Runtime Verification (RV) [2]. RD captures the received input events of the software into a trace. The events are reconstructed from the trace in order to replay the execution sequence. During the replay the behaviour of the software is analysed and debugged. RV checks whether a user-defined Correctness Property (CP) is held by the software. A Checker is synthesized from the specified properties. It checks the conformance of the observed runtime behaviour of the software with the specification. The Checker can run in parallel to the software during the operation [3] or it can analyse comprehensive log traces [4]. In both methods the execution can not be replayed in order to test if the failure still occurs after the developer has debugged the code.


3 Methodology

In our approach the input events of an embedded software are captured during operation and stored into a trace. When a faulty behaviour of the software is detected, the corresponding event sequence is replayed in the laboratory, similar to RD. During replay, the conformance of the runtime behaviour with the specification is automatically checked by applying RV. Violations are indicators for the faulty implementation in the source code which causes the failure. The workflow is illustrated in Figure 1 and each step is described in the following.

A. Instrumentation

Generated code probes are injected into the source code of the software based on an instrumentation specification independent from the underlying platform. The probes write a log of each received event into a trace. These events represent inputs from sensors, from user interactions, or from other external systems. The log for an event includes: type and parameters of the event, system time, and current cycle count for avoiding overlapping timestamps. This way requires less overhead then fine granular capturing for offline log analysis.
B. Runtime Tracing: The test engineers execute the target system including the instrumented software in the operation environment. It writes an execution trace first into a buffer and when the system is idle or a limit is exceeded it incrementally moves the logs into a trace file. When the engineers detect a defective behaviour of the system during operation (e.g., the software shuts down with a segmentation fault message) the current trace file is categorized as failure trace.

C. Trace Transformation: Back in the laboratory, the failure trace is transformed into a test case. It replays the original execution by sending the events to the software in the same way they were sent to the software during operation. The correct temporal order is achieved by setting up a happened-before relation graph for the events (like [5]).

D. Runtime Verification Driven Debugging (RVDD): This step involves the application of RV during replayed execution combined with manual debugging. The specification of the software modelled with the Unified Modelling Language (UML) is reused for the RV. Program-specific definitions for RV Correctness Properties (CPs) are integrated into the models by the developers. Universal CPs can be selected as well (e.g., for concurrency or for memory leaks). The CPs are conditions and invariants which must hold during the execution of the software. They reference variables and parameter values, variable accesses, invocations of functions, as well as timings of operations (e.g., a CP specifies that an array index variable must be between a lower and an upper bound). A Checker module is generated from this specification. The previously generated test case and the Checker are executed with a symbolic debugger connected to the software running on the target hardware. The test case triggers the failure sequence. In parallel, the Checker automatically verifies the conformance of the runtime behaviour of the software with the specification in detail by stepping through the code with the debugger. The Checker reports violations of the specification (e.g., the array index variable is set to a value greater than the upper bound). These are often the causes for a defective behaviour of the software. By achieving the conformance of the software with the specification (e.g., preventing out of bounds value assignments) the error is fixed (e.g., the segmentation fault).

4 Applying RVDD on IVI Platforms

We are currently applying our approach to embedded systems in the automotive area, especially in the field of In-Vehicle Infotainment (IVI). These systems are integrated in modern cars in order to assist the driver by providing context-aware information. Forward-looking for IVI is the GENIVI standard [6] defined by a consortium of key players in the automotive area (e.g., BMW, Bosch, Hyundai, GM). A popular open source GENIVI-compliant operating system is MeeGo IVI 3. It contains the navigation assistance software Navit 4. We have chosen Navit, because testing GPS input with common test frameworks is difficult. In the following paragraph the workflow for the case study is described.

3 is available at https://meego.com/devices/in-vehicle
4 is available at http://www.navit-project.org/
Our tool instruments Navit with the instrumentation tool Clang\textsuperscript{5}. Then Navit is executed during a test drive. When a failure is detected the current trace is stored. Back in the laboratory, Navit is loaded with the GNU Debugger\textsuperscript{6} (GDB) on the MeeGo IVI platform. Therefore, we will enhance our framework for RV with the GDB [7]. A test case script and a Checker script are weaved into a composed script which controls the GDB. The test case script derived from the trace triggers the replay of the events. The Checker script derived from the UML specification observes and verifies the behaviour of Navit by stepping through the code with the GDB. When a violation to the specification is detected, it can be switched to manual debugging. We will define failures which our tool will be able to detect. We will inject these failures into Navit for experiments.

5 Conclusion

This paper presents our recently started work for the optimization of debugging by combining Replay Debugging and Runtime Verification. Input events from external systems to the embedded software are captured during operation into a trace. A failure-causing trace is transformed into a test case which replays the input sequence in the laboratory. Thereby, execution sequences can be replayed without manual trial and error. During the replay, an automatically generated Checker compares the observed behaviour of the software with the requirements of the UML specification. Based on the detected violations the failure can be fixed efficiently. In this way, expensive manual failure analyses are not required.

References


\textsuperscript{5} is available at http://clang.llvm.org/
\textsuperscript{6} is available at https://www.gnu.org/software/gdb/
Debugging as a Service for Localizing Concurrency Faults

Feyzullah Koca\textsuperscript{1,2}, Hasan S"ozer\textsuperscript{2}, Rui Abreu\textsuperscript{3}, and Ismail Ari\textsuperscript{2}

\textsuperscript{1} TUBITAK BILGEM, Information Technologies Institute, Kocaeli, Turkey
\textsuperscript{2} Department of Computer Science, "Ozyegin University, Istanbul, Turkey
\textsuperscript{3} Faculty of Engineering, University of Porto, Portugal
\{feyzullah.koca, hasan.sozer, ismail.ari\}@ozyegin.edu.tr
\textsuperscript{2} rui@computer.org

Abstract. Previously, we introduced a technique and a tool, named SCURF, for debugging concurrent software and identifying faulty code blocks automatically. Concurrency faults are activated by specific thread interleavings at runtime. SCURF systematically instruments the program to create versions that run in particular combinations of thread interleavings. All these versions are subjected to tests and spectrum-based fault localization is used for correlating detected errors with concurrently executing code blocks. However, depending on the code size and the number of threads, resource requirements of SCURF highly vary. In this position paper, we propose debugging as a service (DaaS) for exploiting cloud computing technologies to scale SCURF on demand. Although Testing as a Service (TaaS) has been widely studied in the literature, DaaS has not received much attention. As we propose in this paper, DaaS provides opportunities for the development of more effective and scalable software debugging techniques, especially for concurrent software.

Keywords: fault localization; cloud computing; debugging as a service; concurrency faults

1 Introduction

Concurrency faults are activated by specific thread interleavings at runtime, which makes them hard to detect by testing since they do not deterministically lead to an error. Traditional fault localization techniques and static analysis fall short to detect these faults efficiently. Previously, we introduced a technique and a tool, named SCURF \cite{Koca2015}, for debugging concurrent software. SCURF systematically instruments a multi-threaded subject program to force context switch within different code blocks. As such, each version turns out to be actually the same program that is executed with a different combination of thread interleavings. All these versions are subjected to tests. Spectrum-based fault localization \cite{Koca2016} is used for correlating detected errors with concurrently executing code blocks. The result is a ranking of code blocks with respect to the probability that their re-entrance causes the detected errors.
The number of possible thread interleavings at runtime changes according to code size and complexity. SCURF creates a different version for each possible thread interleaving. Each of these versions are tested with a test suite. Hence, there is a high variation in the number tests to be performed. In fact, these tests can be performed in parallel, even on different machines, if the necessary resources are provisioned.

Cloud computing has emerged as a new computing paradigm that enables ubiquitous access to a shared pool of configurable computing resources. These resources can be rapidly provisioned and released with minimal management effort. Thus, cloud computing technologies offer an opportunity to seamlessly scale SCURF based on demand. In this position paper, we propose debugging as a service (DaaS) that exploits this opportunity. Although Testing as a Service (TaaS) has been widely studied in the literature [3], DaaS has not received much attention. DaaS can facilitate the development of more effective and scalable software debugging techniques, especially for concurrent software.

In the following section, we provide background information on SCURF and cloud computing. Then, we describe our approach for exploiting cloud computing technologies to scale SCURF on demand, and as such introduce DaaS.

2 Background

In this section, we first provide a brief background on spectrum-based fault localization and our tool, SCURF. Then, we discuss cloud computing technologies that will be utilized by our approach together with SCURF.

2.1 Spectrum-based Fault Localization and SCURF

Spectrum-based fault localization (SFL) is a dynamic program analysis technique. The program is executed using a set of test cases and so-called hit spectra is collected, indicating whether a component was involved in a test run or not. Hit spectra is expressed in terms of an activity matrix $A$. An element $a_{ij}$ is equal to 1 if component $j$ took part in the execution of test run $i$, and 0 otherwise. The error vector $e_i$ represents the test outcome. The element $e_i$ is equal to 1 if run $i$ failed, and 0 otherwise. A similarity coefficient is used for measuring the similarity between $e$ and the activity profile vector $A_{ij}$ for each component $j$.

We adapted SFL for localizing concurrency faults. For this purpose, we modified the collected hit spectra and the analysis process used for localizing regular faults. The adapted approach is based on systematically instrumenting the program to trigger a context switch in different components. As such the same program can be tested in different thread interleavings, potentially triggering an error. SFL is applied to reveal the particular thread interleavings of faulty components that lead to the detected errors. Our tool SCURF [4], realizes this approach in 3 steps. In the first step, the program under test is instrumented to generate different versions each of which execute in different thread interleavings. At this step, the program code is instrumented also to collect spectra information
at runtime. Second, each version is tested being subject to the same test suite. Program-spectra are collected for the number of re-entries to each component within a function. Third, the collected spectra are analyzed and correlated with the detected errors. All the components are ranked with respect to the probability that they are subject to a concurrency fault as the cause of the detected errors. These components should be further analyzed by the programmer and possibly considered for introducing thread-safety.

2.2 Cloud Computing and MapReduce

Cloud computing enables ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. Several cloud computing technologies and tools have been introduced within the last decade. For instance, new distributed data processing frameworks such as Apache Hadoop [1] have been invented.

Apache Hadoop is an open-source framework written in Java designed for supporting distributed applications. Hadoop framework mainly consists of two components called MapReduce (MR) and a distributed file system (HDFS). HDFS consists of a NameNode (master) and DataNodes (slaves). MR is a distributed job execution system consisting of a JobTracker (master) and TaskTrackers (slaves). Jobs are split into many Map and Reduce tasks and sent to many machines containing the data in HDFS. JobTracker distributes these client-submitted jobs and TaskTrackers track the progress of local Mappers and Reducers. The Map phase handles data transformations in each worker node and is followed by the sort-merge phase. The Reduce phase aggregates sorted data and outputs the results. In Hadoop version 2.x, HDFS has a namespace federation feature, which provides scalability beyond 10,000s of nodes, since the NameNode is also distributed.

3 Approach

SCURF [4] tries to find schedules that can trigger concurrency faults among potentially a vast number of schedules. To this aim, it forces context switches in different code components. SCURF first creates different versions of the program, each of which is instrumented to trigger a context switch at one of the components. So, if there are \( n \) components as part of the tested function, \( n \) different versions are created and tested. This phase is called as Level 1. Then, SCURF moves on to Level 2, in which context switch is triggered in two different components at each version. So, \( \binom{n}{2} = n \times (n-1)/2 \) different versions are tested. This process continues until the concurrency fault is localized. At Level \( k \), the number of versions to be tested becomes \( n + n \times (n-1)/2 + n \times (n-1) \times (n-2)/6 + ... + n!/k!(n-k)! \). For tractability, the number of levels (i.e., \( k \)) is limited by 2 in the current implementation of SCURF. At Level 2, SCURF takes about
21.3 seconds for a Pentium 4 3.0 GHz HT computer to analyze a function that has 10 components. This duration depends on the program and test inputs, and it increases quadratically with respect to the number of components.

In fact, SCURF performs each test on a different process, which can be executed in parallel. We plan to exploit this feature to increase the scalability of SCURF by distributing its processes on a Hadoop cluster. For instance, if there are 10 components and 3 cluster nodes, while one node is executing tests of Level 1, 4, 7, and 10, the other node can execute tests of Level 2, 5, and 8, the last node can execute tests of Level 3, 6 and 9. Figure 1(a) depicts how SCURF works now on a single computer and Figure 1(b) shows how SCURF would work on a cloud environment. Rectangular boxes represent tests, each of which runs on a different process on the assigned node.

![Central approach and Distributed approach](image)

(a) Central approach
(b) Distributed approach

Fig. 1: Central SCURF approach and proposed distributed approach using Hadoop.

Hadoop was specifically designed for challenging data management problems ("Big Data") and it mainly supports data-intensive distributed Java applications. SCURF, on the other hand, currently developed as a C makefile project. Therefore, we plan to use the Hadoop streaming technology [1]. We plan to utilize MapReduce component of Hadoop just for distributing jobs on different nodes. SCURF does not involve data-intensive computations. Hence, a Mapper function will be sufficient without using any Reducers.

References