Information and Software Technology 96 (2018) 78–93

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A framework for the recovery and visualization of system availability scenarios from execution traces

Jameleddine Hassine*[a], Abdelwahab Hamou-Lhadj[b], Luay Alawneh[c]

[a] Department of Information and Computer Science, King Fahd University of Petroleum and Minerals, Dhahran, KSA
[b] Electrical and Computer Engineering Department, Concordia University, Montréal, Canada
[c] Department of Software Engineering, Jordan University of Science and Technology, Irbid, Jordan

ARTICLE INFO

Keywords:
Non-functional requirements
Dynamic analysis
Availability
Use case maps
Log filtering
Log segmentation

ABSTRACT

Context: Dynamic analysis is typically concerned with the analysis of system functional aspects at run time. However, less work has been devoted to the dynamic analysis of software quality attributes. The recovery of availability scenarios from system execution traces is particularly important for critical systems to verify that the running implementation supports and complies with availability requirements, especially if the source code is not available (e.g., in legacy systems) and after the system has undergone several ad-hoc maintenance tasks.
Objective: Propose a dynamic analysis approach, along with tool support, to recover availability scenarios, from system execution traces running high availability features.
Method: Native execution traces, collected from systems running high availability features, are pre-processed, filtered, merged, and segmented into execution phases. The segmented scenarios are then visualized, at a high level of abstraction, using the ITU-T standard Use Case Maps (UCM) language extended with availability annotations.
Results: The applicability of our proposed approach has been demonstrated by implementing it as a prototype feature within the *jUCMNav* tool and by applying it to four real-world systems running high availability features. Furthermore, we have conducted an empirical study to prove that resulting UCM models improve the understandability of log files that contain high availability features.
Conclusion: We have proposed a framework to filter, merge, segment, and visualize native log traces. The framework presents the following benefits: (1) it offers analysts the flexibility to specify what to include/exclude from an execution trace, (2) it provides a log segmentation method based on the type of information reported in the execution trace, (3) it uses the UCM language to visually describe availability scenarios at a high level of abstraction, and (4) it offers a scalable solution for the visualization problem through the use of the UCM stub-plug-in concept.

1. Introduction

Software comprehension is an essential part of software maintenance. Gaining a sufficient level of understanding of a software system to perform a maintenance task is time consuming and requires studying various software artifacts (e.g., source code, documentation, etc.) [1]. However, in practice most of the existing systems have poor and/or outdated documentation, if it exists at all. One common approach for understanding system behavior is to analyze its run-time behavior, also known as *dynamic analysis* [1].

Dynamic analysis is typically concerned with the analysis of system behavioral aspects at run time [2]. This type of analysis does not require the availability of source code, although it may make use of it (e.g., through instrumentation) when present. A trace or a log is a sequential set of events captured during any particular run of software execution. For example, a trace/log can capture software execution paths, events triggered during software execution, or user activity. In this paper, the terms traces and logs are used interchangeably.

Non-functional requirements (NFRs), e.g., availability, security, etc., are critical to the success of almost every nontrivial software system. While the dynamic analysis of software functional properties has been widely studied, there is much less work covering the dynamic analysis of software quality attributes. The widespread interest in dynamic analysis of software quality attributes provides the major motivation of this research. We, in particular, focus on dynamic analysis of *availability* quality attribute.

* Corresponding author.
E-mail addresses: jhassine@kfupm.edu.sa (J. Hassine), abdelw@ece.concordia.ca (A. Hamou-Lhadj), lmalawneh@just.edu.jo (L. Alawneh).

https://doi.org/10.1016/j.infsof.2017.11.007
Received 9 April 2017; Received in revised form 11 September 2017; Accepted 13 November 2017
Available online 21 November 2017
0950-5849/

Several definitions of *availability* have been proposed [3–6]. According to ANSI [3], the availability of a system may be defined as the degree to which a system or a component is operational and accessible when required for use. The ITU-T recommendation E.800 [5] defines *availability*, as the ability of an item to be in a state to perform a required function at a given instant of time, or at any instant of time within a given time interval, assuming that the external resources, if required, are provided. Avizienis et al. [4] defined the availability of a system as being the readiness for a correct service. Jalote [6] deemed system availability to be built upon the concept of system reliability by adding the notion of recovery, which may be accomplished by fault masking, repair, or component redundancy.

Bass et al. [7] defined a template, for a general availability scenario, composed of six attributes, namely, *Source* (e.g., internal/external to the system), *Stimulus* (e.g., type of fault such as crash, timing, etc.), *Artifact* (e.g., system's processors, communication channels, etc.), *Environment* (e.g., normal operation, degraded mode), *Response* (system should detect event then react to it by performing one or more of the following: record it, notify appropriate parties, disable sources of events that caused the fault/failure, be unavailable for a pre-specified interval, and continue to operate in normal or degraded mode), *Response Measure* (e.g., time interval when the system must be available, repair time, etc.). For example, when an unanticipated external message (i.e., *Stimulus*) is received by a process (i.e., *Artifact*) during normal operation (i.e., *Environment*), the process informs the operator (i.e., *Response*) of the receipt of the message and continues to operate with no downtime (i.e., *Response Measure*) [7]. It is worth noting that not every availability scenario has all of the six attributes [7]. Only the attributes that are necessary to the achievement of the scenario should be specified.

In this research, we define the availability scenario as the sequence of events resulting from the execution of a system high availability (HA) feature. In fact, the triggering of an HA feature within a system or a component (i.e., *Artifact*), starts with the detection of a fault (i.e., *Stimulus*), and leads to a set of events/actions (i.e., *Response*). For example, when a process (i.e., *Artifact*) in Cisco IOS XRv operating system crashes (i.e., *Stimulus*), the process restartability feature respawns (i.e., *Response*) the crashed process. As a result, the process resumes its normal routine. Section 5.1.4 provides a detailed description of this specific scenario.

We, in particular, focus on recovering availability scenarios from system execution traces. This is particularly important for critical systems to verify that the running implementation supports availability requirements, especially if the source code is not available (e.g., in legacy systems) and after the system has undergone several ad-hoc maintenance tasks.

In this paper, we extend and build upon our preliminary work [8] on recovering system availability features from execution traces.

The key contributions of this paper are as follows:

1. It proposes a dynamic analysis framework to recover availability scenarios from system execution traces. The proposed framework does not require the availability of system source code and uses the natively generated log data as a basis for availability scenarios recovery. Our approach proposes a trace abstraction technique that filters and segments a single or many consolidated execution traces into clusters representing execution phases that describe the executed scenario of the high availability feature.
2. It uses the Use Case Maps (UCM) language, part of the ITU-T User Requirements Notation (URN) [9] standard, to visualize the system availability features that have been recovered from the execution traces. The resulting UCM models have been extended with availability annotations (e.g., exception paths, metadata attributes, etc.).
3. It provides a prototype tool that automates the filtering, segmentation, and visualization of the recovered high availability scenarios. Our prototype is implemented as a feature within the jUCMNav [10] tool (see Section 4.4).
4. It demonstrates the feasibility of the proposed approach by applying it to four real-world case studies: (1) Cisco HSRP (Hot Standby Router Protocol) redundancy protocol, (2) Cisco ASA firewall active/standby redundancy feature, (3) Cisco ASR 9000 RSP stateful switchover feature, and (4) Cisco IOS-XRv process restartability.
5. We conducted an empirical validation to prove that the use of the proposed approach improves the understandability of execution traces that contain high availability features. The collected data is statistically analyzed using proven statistical methods, such as *t-test* [11].

The remainder of this paper is organized as follows. The next section presents and discusses the related work. Section 3 introduces briefly the Use Case Maps language and a subset of its availability annotations. Our proposed approach for the recovery of availability scenarios from execution traces is presented in Section 4. In Section 5, we present and discuss the empirical validation of our approach. A discussion of the benefits and limitations of our approach and a presentation of the threats to validity is provided in Section 6. Finally, conclusions and future work are presented in Section 7.

2. Related work

In this section, we survey the literature related to: (1) trace abstraction, (2) trace analysis, with a focus on non-functional requirements, and (3) trace visualization.

2.1. Trace abstraction

Traces tend to be extraordinary large and contain a lot of noise, which often hinders viable analysis (see [12] for a discussion on trace complexity). To address this issue, many trace abstraction and simplification techniques have been proposed with a common objective being to extract high-level views from raw traces (e.g., [13,14]). Cornelissen et al. [14] used a filtering approach for abstracting scenario diagrams by removing events that take place at nesting levels higher than a certain threshold. Kuhn et al. [15] also used a minimal nesting level threshold as one of several filtering criteria to reduce trace size. Hamou-Lhadj et al. [16] proposed a so called *utilityhood* metric to measure the extent to which an event can be considered as a utility or not. They proposed a trace abstraction technique based on the automatic removal of utilities.

Filtering-based techniques have been augmented with pattern detection to better characterize the main content of a trace. Hamou-Lhadj et al. [17], for example, proposed an approach to group similar (but not necessarily identical) sequences of trace events into patterns. Their argument is that users who browse a trace only need to look at the same patterns once. This decreases the time and effort spent exploring the trace. Another approach by the same authors consists of summarizing the content of large traces [18]. The idea is to use static analysis to measure various properties of trace elements (e.g., number of callees, callers, etc.) and rank the trace elements according to their relevance. Trace summaries contain elements that have high ranking.

Other researchers have proposed the use of sampling techniques to reduce the size of traces (e.g., [13,19,20]) just as it is used to reduce data in traditional information theory. Sampling consists of selecting parts of the trace. Chan et al. [20] proposed a random sampling approach in which every *N*th generated event is selected. The problem with this technique is that it might miss important information. It is also hard to generalize. In addition, there is no guarantee that the resulting sample is representative of the original trace. Pirzadeh et al. [13] proposed a rather novel trace sampling approach based on the concept of execution phases [19]. First, the trace is segmented into phases, which can then be used as strata to guide the sampling process. The resulting trace (i.e., the sample) captures all key characteristic of the trace. In this research, we also use trace segmentation in the recovery of