Software Black Box: an Alternative Mechanism for Failure Analysis

Sebastian Elbaum
Computer Science and Engineering Dept.
University of Nebraska, Lincoln
Lincoln, NE 68588-0115
elbaum@cse.unl.edu

John C. Munson
Cylant Technology, LLC
Moscow, ID 83843
johnm@cylant.com

Abstract

Learning from software failures is an essential step towards the development of more reliable software systems and processes. However, as more intricate software systems are developed, determining the nature and causes of a software failure becomes a great challenge. And although many existing techniques can help to understand the nature of the failure, they are limited in some of the following aspects. First, they work only within controlled environments. Second, they have a major impact on the target system behavior. Third, they assume that a failure can be reproduced. Fourth, they lack enough support to carry out a structured failure analysis. In this paper, we present the Software Black Box (SBB) as an alternative mechanism for failure investigation. The SBB is different from its predecessors in that it was specifically designed to be embedded in a target system and assist in the investigation of failures by reconstructing the events that lead to the failure. The SBB architecture is discussed, and a set of failure scenarios that reveal the SBB potential in assisting failure investigation is presented.

1. Introduction

The traditional method to understand failures consists of executing the program until the failure manifests itself. Then, the investigator re-executes the program, stops, examines the program's states, inserts assertions and re-executes the program in order to collect additional information about the failures. Although this cyclic approach has proven to be effective with relatively small systems, with complex systems it is nearly impossible because:

- Continuous operation rules out interactive debugging. The systems cannot be suspended to inspect its states because of their close relationship with their environment, clocks, etc.
- Non-deterministic and non-repeatable nature of real-time systems makes them produce different outputs even under the same inputs.
- Compilation and linkage overhead in large systems is unacceptable.
- Timing constraints rule out intensive intrusive monitoring because the data bandwidth needed might perturb the system behavior.
- Field failures might not provide enough information to even start this process.

To address these problems, researchers have considered various techniques including memory dumps, traces, slices and replay architectures [17, 1, 2, 3, 4, 5, 16]. Memory dumps provide a snapshot of the memory and registers used by the application (.e.g., core dumps in Unix type systems). Traces provide a continuous information flow of a certain type of event [9]. Replay architectures provide an environment to exactly reproduce the activities that led to the failure. Despite the growth of these techniques, the limitations are palpable [14, 15, 18, 7]. Replay architectures and traces are very intrusive and can greatly perturb the application. Memory dumps are benign but they do not provide a history of the events that led to the failures. Furthermore, the presented approaches provide information at a very low level that may not be useful for large systems (e.g., traces of reads and writes), and cannot be plugged-in into a product. Alternative mechanisms for software failure investigation are necessary, especially for complex real-time systems.

The approach we have taken is a bit different. It focuses on the definition of a Software Black Box (SBB)[11]. The key concept behind the SBB is suggested by current practices in aerospace and aviation industry. The SBB is the analog of the Flight Data Recorder: it collects data regarding the main aspects of the software system behavior in order to assist in a post-failure investigation. The SBB constitutes a framework that facilitates the investigation of software failures through two main components. The first component of this framework is the Software Black Box Recorder (SBBR),

which captures a synthesized version of the behavior of an executing system. The second component is the software black box decoder (SBBD). The SBBD generates and quantifies the scenarios that may have led to the failure based on the system structure and the SBBR data. As with the FDRs, the data retrieved from the SBB provides a means for reconstructing the circumstances that led to the failure. Preliminary studies have shown that the SBB overcomes some of the limitations of the other approaches.

2. SBB Foundational Model

The formulation of the SBB failure investigation system is based on the realization that a software system fails only when executing a particular sequence of functionalities [12]. This is obvious when observing systems that run smoothly and reliably for long intervals and suddenly become very unreliable when the software mission changes (e.g. Y2K). It is the execution of a specific set of functionalities, which directly leads to the unreliability [13]. The shift from one sequence of functionalities that executed "good" code to a different sequence of functionalities that exercised unstable code leads to a system failure. The idea behind the SBB is to recreate the sequence of functionalities that were executing prior to the software failure, assuming that this information will be valuable in understanding why the software failed.

2.1. A Behavioral Model

Software systems are designed to fulfill a set of requirements. Some of these requirements are functional requirements, which specify a function that the system or a system component must perform. Each one of these functional requirements or functionalities expresses a user's requirement. The software design process consists of assigning functionalities in F to specific program modules. The design process defines a set of relations, ASSIGNS over $F \times M$ such that ASSIGNS(*f, m*) is true if functionality *f* is expressed in module *m.*

When a program executes a functionality, it will apportion the activities among a set of modules. As the application begins executing, the operating system transfers the execution control to the leading module of the intended application. During software usage, the control of execution passes between the different modules through software calls. A call is a transfer of control from one software module to another with the implication that the control will return to the calling module at some point. The software will apportion its activities among this set of modules and the calls from one module to the next will establish a transition. These transitions constitute observable events.

A recorder could monitor these events and, based on the relationship between functionalities and modules, a decoder could derive the functionalities being executed. It is worth noting that functionalities are not directly observable at execution time, but modules transitions are. The question then is why are we interested in functionalities if we know the modules that led to the failure? The next section answers that question and extends the relationship between functionalities and modules.

2.2. Module Classification

For a given system S, the set M denotes the set of all program modules that constitute S. For each functionality $f \in F$, there exists a relation c over $F \times M$ such that $c(f, m)$ defines the number of functionalities that might be able to execute a given module. Classifying these software modules results in two distinct sets. One set contains the modules associated exclusively with one and only one functionality, that is, the set of uniquely related modules,

$$M_u = \{m : M \mid f \in F, c(\mathrm{f,m}) = 1\}.$$

The second set contains the modules that might be executed by more than one functionality, that is the set of shared modules:

$$M_s = \{m : M \mid f \in F, c(\mathrm{f,m}) > 1\}$$

The existence of this set indicates that knowing which module was executing at the time of the failure is not enough to understand what the software was doing at the time of the failure because a module might execute under different functionalities.

From a different perspective, a relationship can also be established between functionalities and the software modules that they might cause to execute. For each functionality $f \in F$, there is a relation *p* over $F \times M$ such that $p(f, m)$ defines the chances of executing module *m* when the system is executing functionality *f.* Essentially, program modules pertain to one of two mutually exclusive and complementary sets. If $p(f, m) < 1$, then a module *m* may or may not execute when functionality *f* is expressed. The set of potentially involved modules is:

$$M_p^{(f)} = \{m : M_F \mid \exists f \in F, \mathrm{ASSIGNS}(f, m) \wedge 0 < p(f, m) < 1\}$$

Other program modules exhibit a tight binding with a particular functionality. These modules are said to be indispensably involved with the functionality *f.* This set of indispensably involved modules for a particular functionality *f* pertain to the set of those modules with the following property:

$$M_i^{(f)} = \{m : M_F \mid \forall f \in F, \mathrm{ASSIGNS}(f, m) \wedge p(f, m) = 1\}$$

With these definitions, we can formalize the concept of a functionality in terms of modules. A functionality has to

have at least two modules, and at least one of them must be an element of M_u and M_i. That is, $f_i \in F$ if

$$f_i = \{ \forall m \in M^{f_i}, \exists m_j \in M_u \wedge m_j \in M_i, \left\| M^{f_i} \right\| > 1\}$$

Then, a functionality constitutes a higher level abstraction that provides a context for a given set of modules.

3. SBB Architecture

The goal of the SBB is to assist in the investigation of failures. Failure investigation can be performed in two different contexts: (1) within a controlled development environment, in which case it is closely related to the testing, fault reproduction and debugging activities, or (2) it can be performed in the field, when a system has already been released. Many of the tools mentioned in the introduction can assist in the investigation of failures within a controlled environment. The SBB design on the other hand, was produced with the second scenario in mind, where a system is shipped and it is necessary or convenient to have a device that would assist in the understanding of field failures. This device has to minimize the perturbance of the target system and be robust enough to survive a failure. The design of the SBB reflects that decision. A second but related design decision is the separation of the recording mechanism, SBBR, from the interpretation mechanism, SBBD. The SBBR records the behavior of the target system. When the system fails, the SBBD analyzes the recorded information. There is not interaction between these two components before the software failure. This approach reduces the size of the recording device minimizing the perturbance on the target system. The next two sections describe the architecture and tradeoffs of the SBBR and the SBBD.

3.1. Software Black Box Recorder

The SBBR is the recording device where the system data is stored. The major challenge of the SBBR is to capture the essential characteristics of the system behavior to provide useful information in case of a failure, while minimizing the perturbation. This section presents the model, the architecture and the primary characteristics of the SBBR.

3.1.1. Measuring Transition Information. It has been established that system transitions from one module to another are observable and can be measured. The transitions among modules can be represented through an *n* by *n* adjacency matrix, where *n* represents the number of modules in the system. The value of each cell in the matrix will equal True (T) if there exists a call from m_i to m_j. A slightly variation of the adjacency matrix can also account for associated frequency. As each module is called, the transition to that module can be recorded in the matrix by keeping count of how many times a certain module was exercised. After each transition, an element in the matrix is incremented. This describes a particular kind of matrix called a frequency matrix. Derived from this matrix, a second matrix can be generated to contain the different transition probabilities. This new matrix, the transition probability matrix, represents the probability p_{ij} of transitioning from m_i to m_j.

More formally, let $m_{\bullet j} = \sum m_{ij}$ represent the row marginal for the jth module and $m_{\bullet} = \sum m_{\bullet j}$ the total number of observed transitions. Point estimates may be derived from the steady state [12] system activity represented by the SBBR frequency matrix as follows: $p_{ij} = m_{ij} / m_{\bullet j}$.

Although the matrices presented above seem useful in the representation of the system behavior, they provide no information regarding the order in which the epochs occurred. Sequences are very useful in this regard. A sequence is an ordered set of events. For example, a software system during execution might generate a sequence of module execution events represented by $(m_1, m_2, m_3, m_4, m_5, \ldots)$ where m_i represents the execution of a module. A new element is added to the sequence every time a new transition is made into a module.

3.1.2 Recorder Basic Architecture. The proposed SBBR has three major components: a front end, a kernel and a back end. The front end receives and validates the input data received from the target system. The kernel contains three primary data structures aim at the transition measurement: the transition sequence, the frequency matrix and an internal call-stack used to assist in the updates of the first two data structures. The transition sequence is stored in a queue that contains the last *n* transitions in the sequence. With this scheme, the previous transitions disappear and the sequence only reflects the latest software behavior. The transition frequency matrix complements the transition sequence by reflecting the behavior of the system since the beginning of execution. From the transition frequency matrix a transition probability matrix can be derived. This probability matrix represents the program behavior in terms of the likelihood of the different paths that the system could have taken. The back end dispatches the data stored in the SBBR to safety in the case of a failure. In addition, the back end of the SBBR allows the operation and control of the recorder. Our implementation of the SBBR allows the use of interrupts to control the SBBR activation/deactivation, download data, re-initialize and resize the sequence size. Further details about the architecture can be found in [7].

3.1.3. Recorder Survivability. The data contained in the SBBR needs to survive the failure in order to be valuable. The simplest method to ensure the survivability of the SBBR is transferring the content of the recorder to a non-volatile storage device. This transfer can occur periodically, or triggered by a specific event. If the chosen transfer intervals are small, little or no information loss occurs during the failure. However, the performance of the target system suffers. On the other hand, triggering the data transfer less frequently decreases the impact on performance. Unfortunately, a system failure may lead to a significant amount of data loss. The implementation of the trigger and the data transfer mechanism and its associated trade-offs constitute the key to the SBBR survivability.

Many software mechanisms may be used to provide the necessary survivability capability. We have experimented with exception handlers, parallel processes, remote processes and operating system failure handlers. Unfortunately, none of these mechanisms work perfectly in all situations. The system's environment, the software architecture and the nature of the failures are some of the factors to take into consideration when analyzing the mechanism that will ensure the survivability of the SBBR.

3.1.4. Target System Instrumentation. In order to access the recorder, the target system code must include SBBR calls. This process of inserting "hooks" (snippets of code) into the target system is called instrumentation. For the purposes of this study, we used an instrumentation tool called CLIC [6]. CLIC works only with transitions, so the size of the code instrumented with CLIC increases only slightly. CLIC inserts calls to the SBBR at the beginning and at every exit point of each software module. Once the code has been instrumented, and the SBBR attached, the system can be re-built. This simple procedure embeds the SBBR into the target system.

3.1.5. SBBR Calibration. The calibration process initializes and sets up the SBBR to maximize its effectiveness. There are two aspects of the SBBR that have to be defined through a calibration process:

(1) Sequence Length Definition: there is a trade-off between the sequence size and the amount of information that is required by the SBBR to be useful. After experimenting with several approaches, it was clear that the size of the sequence should be proportional to the size of the system. More precisely, it should be a function of the variance in the behavior of the system. To determine the sequence size, it is necessary to understand the variability of the transition space. If the target system presents large variation in its behavior, more data is necessary to determine its distribution with a certain level of confidence. Based on these ideas, it was determined that a calibration period was needed in order to define the transition space and its distribution. At the beginning of this calibration period, the target application is executed with the embedded SBBR and a sequence of size $n = 1$. The purpose of this initial calibration is to understand how the system distributes its activity across the modules, and what is the variance of this distribution. Based on the variance of the distribution, the size of the sequence is estimated using

$$\delta = \sum_{i=1}^{n} \frac{\left(f_i^{seq} - m_{\bullet} p_i\right)^2}{m_{\bullet} p_i}$$

where the sample size n is incremented until the system provides the same variance across consecutive samples. When the equation provides a set of consecutive variances with the same value, then the size, n, is fixed and becomes the target size of the sequence[1].

(2) Transition matrix initialization: the data in the transition matrix is used to determine the normality of the execution sequence (to be explained in the next section) by providing an estimate of the system long-term behavior. If the system fails in its initial stages, and the transition matrix is not initialized with enough data, the long-term behavior of the system is not reflected in the matrix, and any scenario is likely to be abnormal. To solve this problem is necessary to provide an initial load for the matrix. This initial load should constitute an average normal behavior, and it would prevent the system from a biased assessment due to the lack of enough data.

3.2. Software Black Box Decoder

When a system fails and the SBBR is recovered, the reconstruction stage starts. The SBBD includes a series of techniques that allow the understanding of the data in the recorder. The decoding process generates possible functional scenarios, quantifies the normality of the data and allows further exploratory analysis[2].

3.2.1. Generating Functional Scenarios. The decoding process has to "make sense" out of incomplete and usually ambiguous data, which makes the decoding process non-deterministic. The first source of ambiguity emerges from the poor mapping of functionalities to modules. The modules from M_s, which contains the modules common

---

[1] The sequence size obtained through this procedure is meant to be a guideline. Specific needs of the target system might alter the size estimation provided by this procedure.

[2] Due to space reasons, we have not included the formal algorithms and the exploratory procedures in this paper. For this information please refer to [Elbaum99] and [Elbaum00]

to many functionalities, make the decoding process harder because they cannot be mapped to one functionality. The second source of ambiguity spans from the recording process. If the failure occurs during the execution of a shared module, the functionality that is currently executing might not be clearly expressed.

The algorithm to generate the functional scenarios uses the data provided by the recorder and the mapping between functionalities and modules. An example system *S,* with a call graph representation shown in Figure 1, contains 3 functionalities and 7 modules. The mapping of functionalities to modules is shown in Table 1.

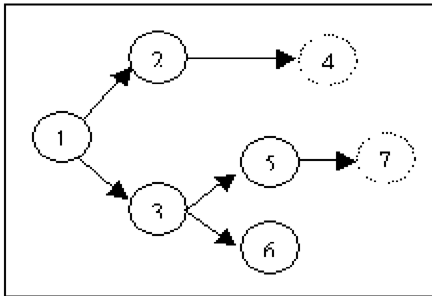


Figure 1. Call Graph

Table 1 Functionalities and Modules

	M1	M2	M3	M4	M5	M6	M7
F1	T	T		T			
F2	T		T		T		T
F3	T		T			T	

The generation process attempts to map the modules in the transition sequence to functionalities. The idea is to extract what functionality might be represented in each one of the transition sequence events. The mapping of each module found in the sequence to a functionality, produces a sequence of functionalities. The algorithm is synthesize in 4 stages:

A. Mapping of M_u. Mapping the modules belonging to M_u presents no difficulty and is direct. Assume that the recorder provides the following transition sequence for system S: ⟨1,2,1,3,5,7,5,3,1,3,6,F⟩. After the transition to module six, the system failed and the recorder stopped. At this stage the reconstruction would look like the following:

Module Sequence	1	2	1	3	5	7	5	3	1	3	6
Functionality Sequence	?	1	?	?	2	2	2	?	?	?	3

Not all the modules belong to M_u. The functionality sequence at this point might contain some sequence cells with no elements in it. These empty cells constitute gaps in the functionality sequence. The gaps break down into three main groups: M-Gap (middle gaps) involves the gaps that are located between known functionalities. R-Gap (right gap) contains only one element and is located to the right of the last identified functionality. L-Gap also contains only one element and is located to the left of the first identified functionality.

B. Removing M-Gap. Many gaps belonging to M-Gap might exist, but they are irrelevant. The definition of functionality guarantees that each functionality must contain at least one unique and indispensable module. Therefore, no new functionalities can reside in this type of gap. Having a new functionality hidden in a gap of this type would contradict the definition of functionality. The modules contained in these gaps belong to either the functionality on the right or the one on the left of the gap.

In view of the fact that they do not provide new information from a functional scenario perspective, the functional scenario generation can ignore the modules from M-Gap. In the previous example, the functional sequence reduces to:

?	1	-	-	2	2	2	-	-	-	3

C. Removing R-Gap & L-Gap. Under a different failure scenario, suppose that the recorder obtained a new sequence ⟨6,3,1,3,5,3,1,3⟩ from S. The decoding process is now more complex. The last transition ⟨1,3⟩ includes modules shared by two functionalities. Unfortunately, no deterministic procedure exists to identify which functionality was going to be executed. Under these circumstances it is possible to generate all possible functional scenarios that might be defined by the modules in the gap belonging to R-Gap. A new structure C_i holds the candidate functionalities for the ith cell in the functionality sequence. After this stage, the functional scenario for ⟨6,3,1,3,5,3,1,3 ⟩ will look like this:

Module Sequence	6	3	1	3	5	3	1	3
Functionality Sequence	3	-	-	-	2	-	↓	↓
Candidate Functionalities							1	2
							2	3
							3	

At this stage there are multiple possible functional scenarios. Not all of these scenarios are in the feasible set of scenarios. The modules involved in the R-Gap can represent at the most two different functionalities, the first one being the right-most identified functionality. Since no other unique module resides within those modules in the gap, representation of no more than two functionalities can

take place. A new functionality can only be found if the first functionality after the last identified functionality is different from it.

D. Scenarios Simplification. Based on this new insight, some of the generated scenarios undergo removal and only the feasible ones are kept. A new structure PS_j (Possible Scenario) holds $\prod size(C_i)$ possible scenarios. In the previous example, the number of possible scenarios equals 6. From the previous C_i, the PS_j in Table 2 can be generated.

For the given example, the last identified functionality equals two. The functional scenarios PS-1, PS-3 and PS-4 introduce two different functionalities, with the first one different from the last identified functionality, which is a contradiction. Therefore, they are not feasible scenarios. The functional scenarios PS-2. PS-5 and PS-6 are feasible based on the algorithm.

Last, by joining consecutive instances of the same functionality into one, the scenarios are reduced. This refinement would make some scenarios equivalent[3]. In the example, scenarios PS-2 and PS-6 will generate the same functionality sequence after reducing them, and they are considered equivalent. The L-Gap of the generated functional sequence can be decoded using the same procedure as the one on the right, but in the opposite direction. For simplicity and without loss in generality, the left gap will be ignored in this paper.

3.2.2. Evaluation of System Normality[4]. The SBBR data can assist not only in the functional scenario generation but also in the analysis of whether the behavior of the system in the last epochs of execution was abnormal. The SBBD can aid in this process by determining whether the transition sequence observed before the failure equals to the steady state of the system.

If the sequence recorded by the SBBR does not fit the steady behavior of the system, then the recorder sequence contains abnormal behavior aiding the interpretation. This would be analog to the analysis of flight data recorder information. If the recorder data does not fit the normal profile of an airplane behavior, then some hypothesis such as human error are more likely to be considered as the cause of the accident.

[3] This procedure might overlook the fact that the repeated execution of the same functionality might be informing. However, we discovered that this simplification is necessary to facilitate implementation when dealing with larger sequences.

[4] By normality we mean nominal behavior. We are not referring to statistical normality.

The procedure to obtain a transition probability matrix from the frequency matrix provided by the SBBR has been defined. This matrix represents the steady state activity of the system since the SBBR began execution. The transition sequence reflects the last *n* epochs of the system execution before the failure. From this sequence, a new frequency matrix may be derived that would represent the activity of the system during the last *n* epochs. The question then reduces to the evaluation of whether the data in the new frequency matrix was extracted from the same population as the steady behavior data. This conjecture may be tested through

where χ^2_γ represents the 100% point of the chi-square distribution with *n-1* degrees of freedom.

$$\sum_{i=1}^{n}\frac{\left(f_i^{seq}-m_\bullet p_i\right)^2}{m_\bullet p_i} < \chi^2_\gamma$$

4. Study

The objective of this study was to present the SBB with a concrete example, and to investigate the potential of the SBB when applied to a target system. Our hypothesis was that the SBB could provide useful information in the understanding of failures not available through other mechanisms. The study was carried within a controlled environment that we proceed to describe in the following subsections.

Table 2 Feasible Scenarios

Corresponding Modules	M1	M3	Extensions	Collapsed
~~PS 1~~	~~1~~	~~2~~		
PS 2	2	3	3-2-2-3	3-2-3
~~PS 3~~	~~3~~	~~2~~		
~~PS 4~~	~~1~~	~~3~~		
PS 5	2	2	3-2-2-2	3-2
PS 6	3	3	3-2-3-3	3-2-3

Target Application. The chosen target application was the popular program suite used to transfer files across networks with the File Transfer Protocol. It is composed of a client that makes requests, and a server daemon that attends those requests. In general, the client interprets user commands, packages the command, communicates it to the local ftp server and waits to hear from the server. The server analyzes the request and responds. The focus of this experiment is specifically on the client side. The flavor of the chosen application works in the Linux operating system and it has approximately 10KLOC and 177 modules. It works with sockets and interacts with daemons, making it complicated enough to be worth

investigating. The FTP source code is publicly available making this study easily reproducible and there is a "list of known bugs", which helped us to generate the failure scenarios.

Mapping Functionalities and Modules. The decoding process is based on the mapping of functionalities to modules. This mapping was performed as a reengineering task. The first mapping draft was generated based on the limited documentation publicly available. A set of candidate functionalities emerged from this initial stage. Then, successive iterations redefined and refined the functionalities and the mapping, based on the definition of functionality from Section 2.2. For some of those iterations, we required the assistance of static tools such as call tree generators, and dynamic tools such as profilers provided by the executing environment. The functionalities obtained by this process and their corresponding modules follow (the module in italics in the indispensable one):

1. "Commander": command validation and handling.
   Unique: *162*,12,49,159,160,163,164,165,166
2. "Set Environment": set the application general variables.
   Unique: *31*,33,34,38,37,40,41,69
3. "Manage Connection": establish, synchronize and keep connection.
   Unique:*103*,13,61,62,78,99,101,102,104,114,117,119,161,32,42,43,44,45,46,47,48,51,52,53,54,55,56,57,58,59,60,77,79,82,83,84,114
4. "Translate": translate naming conventions.
   Unique:*65*,30,128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,152,153,154
5. "Set Transfer": set file transfer modes.
   Unique: *16*,14,15,17,18,19,20,21,22,35,36,40,70,71,73,75,76,81
6. "Authenticate": get name, password and login in remote machine.
   Unique: *100*,50, 60,175,176
7. "Operate Macro": define and execute macros.
   Unique: *86*,80
8. "Transfer to": transfer a file to a remote destination.
   Unique: *106*, 23, 24
9. "Transfer from": transfer a file from a remote destination.
   a. Unique: *108*, 25,26,27,29,85,107,118
   b. Shared: 63,74,109,110,111,112,113,115
10. "Set Proxy": operate in a secondary control connection.
   a. Unique: *68*,67,116
   b. Shared: 63,74,109,110,111,112,113,115

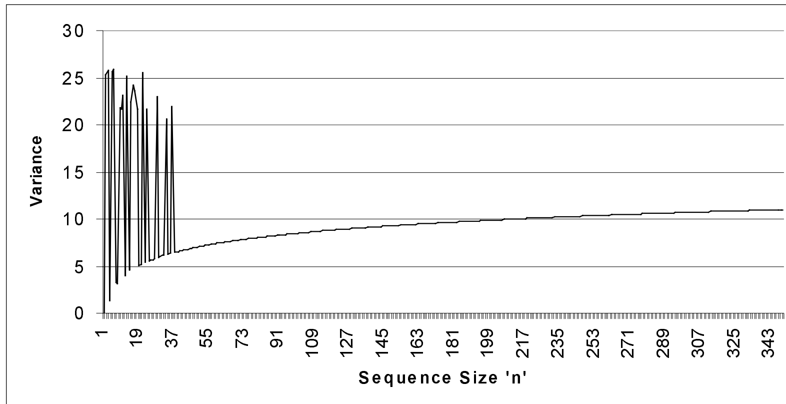
Since no design documentation was available, our mapping and classification were only estimates, where errors were likely to occur. For example, a module that was always executed except for extremely rare conditions may have appeared to be indispensable while it was not. The problems that arise from these uncertainties started disappearing as the software got executed and more of its behavior was known (the probabilities of finding new special situations become more rare) [10].

SBBR Survivability. The SBBR was implemented in C and took advantage of certain constructs within the C language that allow catching errors before exiting the program. These constructs, called exception handlers, acted as the SBBR trigger mechanisms. The SBBR transferred its contents to the local hard drive before the program finished, independently of the exit status.

Embedding the SBBR into FTP. The SBBR was embedded in the target application in two steps. First, the preprocessor provided by CLIC [6] inserted "hooks" (snippets of code) into the FTP modules. These hooks were function-calls to the SBBR. The CLIC preprocessor instrumented a total of 177 modules in 5 files. The second step was the linkage of the SBBR to the target application done primarily through simple additions to the original FTP makefile, that were meant to incorporate the SBBR modules to the target application.

SBBR-FTP calibration. Graph 1 presents the plot of sequences with different size, *n,* and the corresponding observed variance, δ, for the target FTP application. Initially, the sequence size was set to 1. Then, the sequence size was incremented. After oscillating for sequences of size, *n,* below 50, the variance obtained from sequences of different sizes over 50 did not change substantially. The lower threshold for the sequence size should be set after the oscillation period. For the FTP application, a sequence of size 500 was selected. At that point, the incremental gains of having hundreds of extra transitions recorded in the sequence were minimum. In order to determine the sequence size for FTP, over 300K

transitions were needed, that is, over one megabyte of transition information[5]. It can be expected that larger systems, with larger behavioral variance, will need more execution to stabilize, generating a larger amount of data.



Graph 1 FTP Calibration

Before running each test scenario, the target application was put through a regular battery of tests to exercise some normal operations and establish a minimal operational baseline before the failure.

Failure Scenarios. The planned failures are meant to illustrate how the SBB would behave under different circumstances. By exercising the failures, the analysis of the benefits and the limitations of the SBB can be performed. In addition, the failures help assessing what factors are affecting the SBB efficiency. Some of the failures scenarios were extracted from a previously documented list of known problems. This list provides a set of known circumstances that make the product fail. Other scenarios were designed specifically to display aspects of the SBB not visible through the known failures. For all the scenarios, the FTP application was run from Linux. The sessions were opened to a local terminal and several remote terminals that operated HP-UX, Windows NT and DOS. The following is a synthesized description of the planned failures:

1) "Get Fat": When files are retrieved from a remote system using a file naming convention of 8 characters, FTP transforms the file name to fit those restrictions by renaming the files using 6 characters plus an index. It was known that the application had a limitation if more than 99 files were transferred because it exceeded the index capabilities and it produced a buffer overflow.
2) "Chmod": The feature "chmod" was not documented in the target application version. It was included in the code for future release and for "beta-testing". After studying the source code that implemented "chmod", it was clear that the feature had insufficient error control. The "chmod" feature was first exercised over an existing file and then over a non-existing file, where it failed.
3) "Shell Debug". The shell facility was not very stable in the target release. If the shell was used, the debug feature should not be activated. In spite of this, the shell works well if debug is not activated. The shell was used successfully in repeated occasions with the debug option deactivated. Next, the debug option was activated and the attempt of using the shell subsequently produced a failure.
4) "Kill FTPApp". Under this scenario, the FTP application was initiated, a few commands were exercised, and then the application was killed from the operating system.
5) "Remote Info". An attempt to perform any remote operation with no connection in place usually leads to an error message within the application. The only exception to this is the invocation of the "system" command. This feature is supposed to provide information about the remote operating system. Instead, it produces an application failure if a connection has not been previously established.
6) "Kill FTPd". The FTP application works by communicating through a port with an FTP daemon. For this scenario, the daemon was killed while doing a file transfer. Under normal circumstances, FTP stops the file transfer and reports an error. The objective of this scenario was to analyze the behavior of the application when the FTP server goes down.
7) "Idle". The *idle* command allows setting an inactivity timer on the remote connection. If no activity is detected in a specified period of time the connection is closed. The module implementing the command includes a portion of in-line assembly code. The assembly lines are calls to a specific library. The objective of this scenario was to analyze the data provided by the SBBR if the software fails when executing a section of code that invokes a particular library or language. In order to accomplish the scenario objective, a fault was embedded within the assembly code to cause an immediate failure.
8) "Abort transfer". In the sixth scenario, the transfer of files was interrupted by killing the FTP daemon. For this scenario, the transfer of files was interrupted

[5] This technique has also been applied to xv (+800 modules), matrix library (12 modules), real-time system (+1600 modules) and chess game (33 modules).

through CRTL_C in the client connection, and the data from the SBBR was downloaded immediately after the signal was received by FTP simulating a fault in the abort transfer procedure. The objective of this scenario was to investigate a more complex failure scenario where, all the decoding steps are necessary.

9) "Put Sync". When the FTP protocol of communication is broken between the FTP daemon server and the client application, the application attempts to resynchronize itself with the server; if it cannot, the transfer is aborted. If the transfer is aborted, a manual synchronization through the "reset" command can be tried. If successful, the transfer is continued from where it left with the corresponding savings. If a posterior transfer to a remote site is executed after a "reset" command, the application dies. For instance, if a "put" command is executed after a "reset", then the application fails.
10) "Proxy". The "proxy" command executes a command on a secondary control connection to allow simultaneous connections to more than one remote server. For this scenario, a connection was established with a remote server X and then, using the proxy command, a connection was opened from X to sever Y and a file was transferred from Y to X. The implementation of the proxy command in this release fails if the primary connection, in our case X, does not provide the FTP services that the user required.

Evaluation Criterion. For each scenario, the usefulness of the SBBR was qualitatively estimated using the following scheme:

- **Low**: the SBBR triggers are not activated by the type of failure. As a result, no data is gathered. The only lesson that could be learned is that the failure occurred at a level not handled by the SBBR.
- **Medium**: the SBBR survived and provides data. The decoding process found the possible scenario or scenarios that led to the failure. The sequence was considered normal, and provided no additional hint regarding the nature of the failure. The failure may have been caused by a condition or state of the system, or by an external component not observable by the SBBR. This is equivalent to the information provided by a "functional trace".
- **High**: like Medium except that the sequence was considered abnormal indicating that the anomaly that caused the failure was likely to be reflected in the SBBR sequence.
- **Outstanding**: like High but in addition, the generated functional scenarios were likely to explain the sequence of events that led to the failure, and further assist in the reproduction of the failure.

Additional criteria have been established and more could be established. For example, one criterion could be target system perturbance. In [7] it was shown that the SBBR had a lesser impact on the target system performance than core dumps, traces and a replay-architecture. Another criterion could be the usability as a plugged-in device. In [7] it was also shown that the SBBR is as practical as the best memory transfers provided by certain compilers [9]. Last, we could have compared the capability of reconstructing the circumstances that led to the software failure. The replay architectures and certain traces provide this type of information, generally at a very low level, while the SBB provides it at higher level that reaches up to design abstractions. Due to space limitations, we'll report only the results based on the presented criterion.

5. Results

In this section, the information obtained with the SBB is presented. For each scenario, we have included a short description of the findings.

(1) "Get Fat". In the generated failure scenario, the following three functionalities appeared consistently and cyclically: *Manage Connection (3)*, *Transfer From (9)* and *Translate (4)*. The last module to be executed was mapped to the *Translate (4)* functionality, which indicates that this functionality is likely to be associated with the failure. The repetition of these functionalities in the sequence also indicates that the same set of operations was executed in repeated occasions. When analyzed, the failure scenario was considered abnormal mainly because the functional sequence repeated itself, which did not match statistically with the probabilities extracted from the SBBR matrix. In addition, the number of *get* operations was extremely high.

(2) "Chmod". At the time of the failure, the *Manage Connection (3)* functionality was executing, more precisely the *do_chmod (57)* module. In this scenario, knowing the functionality did not provide significant help. The failure occurred only when *do_chmod (57)* executed over a non-existing file. Since these circumstances are not captured by the SBB, it cannot help with the exact reproduction of the failure. The sequence was determined to be abnormal mainly because of the execution of the *do_chmod (57)* module, which was never executed during calibration.

(3) "Shell Debug". The last executing functionality was *Commander (1)* and the last module was s*hell(49)*. The functionality *Commander (1)* is the most common of all, so it was not of much assistance[6]. The SBB provide another interesting piece of information about the scenario in terms of the transitions that led to the failure: the module *set-debug(41)* was executed prior to the *Shell(49)*. In addition, the scenario was considered abnormal, mainly because of the execution of the module *set-debug(41)*.

(4) "Kill FtpApp". There was no data available from the SBBR, which indicates that the failure occurred at a level not being handled by the SBB. The SBBR triggers were implemented at the language level as exception handlers; when the process was externally killed, the handlers were not able to catch the signals sent to the process and the data contained within the SBBR was lost.

(5) "Remote Info". The functionality *Manage Connection (3)* was executing at the time of the failure. Similarly to the "Shell Debug" scenario, the mapping was not optimal for the analysis of this failure. This functionality performs so many different activities, that knowing the functionality is not of much assistance for this scenario. The module executing at the time of the failure was *syst(79)*. The scenario was considered normal, which indicates that there is some special circumstance for the system to fail that is not being handled by the SBB.

(6) "Kill Ftpd". The decoder generated two possible functional sequences that included: *Connection (3)*, *Transfer From(9)*, *Manage Connection(3)*, *Transfer From(9)*, and either *Transfer From(9)* or *Transfer To(8)*. The ambiguity was due to the presence of shared modules in the execution sequence. The last executed module belonged to the functionalities *Transfer From(9)* or *Transfer To(8)*. We then proceeded to artificially extend those sequences until unique modules were found, which led us to conclude that *Transfer From(9)* was the last executing functionality. Since the failure scenario was considered normal, the SBB usefulness qualifies as medium. However, the SBB provided a functional context for the failure that could not have been provided by a typical trace.

(7) "Idle". The last executed functionality was *Manage Connection (3)* and the last executing module was *idle_cmd(59)*. The SBB did not provide any additional assistance, although it could have with the instrumentation of the assembly library. The scenario was considered abnormal because the "idle" command is not a normal activity as specified by the SBBR transition matrix.

(8) "Abort Transfer". The system failed when executing the functionality *Transfer From(9)* or *Transfer To(8)*. The last executing module was *ptransfer(111)*, shared by those functionalities. The scenario was considered normal. The sequence extension was attempted without success because the *ptransfer* module only calls other shared modules, which indicates that the system remained in the last executed functionality: *Transfer From(9)*. It was also noticeable that module *abortrecv(107)*, in charge of handling transfer interruptions in conjunction with *ptransfer(111)*, was executed previously. A different mapping scheme that included these modules in their own functionality, would have provided further assistance by isolating the failure in one focused functionality.

(9) "Put Sync". The decoding procedure generated the same functionalities than the previous scenario except that the last functionality was *Transfer To(8)*. The scenario was considered normal. What was interesting about this scenario was the fact that the fault location was not the same as the failure location. The program fault was located in *put(23),* but the system failed when executing *tvsub(112)*. These modules are part of the decoded functionality that gives a context to the failure. This highlighted one of the advantages of focusing on functionalities. Module information by itself would not have revealed the context of the failure.

(10) "Proxy". At the time of the failure, the module *doproxy(68)* within the functionality *Proxy* was executing. It was interesting to discover there was not any transfer functionality involved, the proxy was performed through commands sent to remote demons, and the *Manage Connection(3)* functionality was used extensively. From the user perspective, this may have looked like a transfer. From the SBB perspective, this is a different and abnormal sequence of functionalities.

---

[6] The lack of a specific "Shell" functionality negatively affected the understanding of this scenario. Please refer to [Elbaum99] for additional observations regarding the impact of the mapping and functionality arrangements.

Table 3 summarizes the results of the failure scenarios. The last column classifies the SBB performance.

Table 3. Summary of the failure scenarios

Failure Scenario (FS)			Sequence Normality	Usefulness
Number	Name	Peculiarity		
1	Get-Fat	Long Repeating Sequence	A	Outstanding
2	Chmod	Untested Feature	A	High
3	Shell-Debug	Global State	A	High-Outstanding
4	Kill FTPApp	External Failure	-	Low
5	Remote Info	Global State	N	Medium
6	Kill FTPd	Multiple Scenarios	N	Medium-High
7	Idle	Different Language & External Library	A	Medium-High
8	Abort Transfer	Multiple Scenarios	N	Medium
9	Put-Sync	Fault Propagation	N	Medium
10	Proxy	New operation	A	Outstanding

6. Lessons Learned

For certain scenarios, the SBB provided unique information that could have not being gathered through other mechanisms. For other scenarios, the SBB did not provide additional assistance. We found that there were several factors impacting the difference in the SBB effectiveness. First, mapping functionalitites to modules. The mapping process has many degrees of freedom and is generally not well documented. For certain failure scenarios, we discovered that different mapping granularity would have helped. For others, a complete different association of modules would have been better. The mapping scheme has a major effect on the SBB usefulness. Second, the type of failure. Some failures were evident in the data collected by our model, while other failures were not even recorded in the SBBR. Third, the architecture of the target system. We estimate that greater modularity, greater fault isolation, cut on global dependencies and enhanced coupling could improve the SBB effectiveness by reducing the ambiguity present in the decoding process. Fourth, target system general characteristics. The study helped us realize that large and complex systems may constitute the most likely targets. The SBB will be more effective on systems where alternative tools cannot deal with the magnitude of the problem, the perturbance, and when a functional understanding becomes necessary.

We plan to carry out further studies on those factors affecting the SBB effectiveness and enhance its capabilities. Preliminary studies indicate that we could use the flexibility of the mapping from functionality to modules to perform more powerful mappings based on the nature of the failure. We have begun to define the architectural variables affecting the SBB, so we can determine for which kind of systems it would be more appropriate. We have also begun to analyze the extensions necessary for the SBB to work in a distributed environment. Finally, we recognize that the SBB is a valuable alternative to understand and software failures, but it is still in its infancy. We are planning to perform additional studies and experiments to further validate our ideas.

References

[1] H. Agrawal and J.R.Horgan, "Dynamic Program Slicing", Proceedings of the ACM SIGPLAN90, June 1990, pages:246-256.

[2] J.M.Anderson, L.M.Berc, et all. "Continuous Profiling: Where have all the cycles gone", ACM Transactions on Computer Systems, V15, N4, November 1997, pages:357-390.

[3] P.Argade, D.K.Charles and C.Taylor., "A technique for monitoring run-time dynamics of an operating system", ACM SIGPLAN on the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1994, Pages:122-131.

[4] T.Ball and J.R.Laurus, "Optimally profiling and tracing programs", Proceedings on the 19th Annual Symposium on Principles of Programming, August 1992, pages:59-70.

[5] J.D.Choi and J.M.Stone, "Balancing Run-Time and Replay Costs in a Trace and Replay System", ACM SIGPLAN on the Workshop on Parallel and Distributed Debugging, V26, N12, November 1991, pages: 26-35.

[6] S. Elbaum and J. Munson, "CLIC: Understanding Program Dynamics", TR-98-02. Computer Science Department. University of Idaho. February 1998.

[7] S. Elbaum, "Conceptual Framework for a Software Black Box". Dissertation. Computer Science Department. University of Idaho. July 1999.

[8] S. Elbaum and J. Munson, "Investigating Software Failures with a Software Black Box", accepted for presentation and publication at the IEEE Aerospace Conference.

[9] S.L.Graham and M.K.McKusick, “Gprof: a call graph execution profiler”, ACM SIGPLAN, V17, N6, June 1982, pages:120-126.

[10] G. Hall, “Usage Patterns: Extracting System Functionality from Observed Profiles”. Dissertation. Computer Science Department. University of Idaho. April 1997.

[11] J. Munson, “A Software Black Box Recorder”, Proceedings of the 1996 IEEE Aerospace Conference. Aspen, Colorado. February 1996.

[12] J. Munson, “A Functional Approach to Software Reliability Modeling”, Quality of Numerical Software, Assessment and Enhancement, Chapman & Hill. London, England, 1997.

[13] J. Munson and S. Elbaum, “Software Reliability as a Function of the Executed Patterns”, Hawaii International Conference on System and Science. January 1999.

[14] R.H.Netzer and M.H.Weaver, “Optimal tracing and incremental execution for debugging long running programs”, TR-CS9411, Department of Computer Science, Brown University, March 1994.

[15] N.Price, “New techniques for replay debugging”, TR-CS9800, Department of Computer Science, Brown University, May1998.

[16] M.W.Shapiro, “RDB: A system for incremental replay debugging”, TR-CS9712, Department of Computer Science, Brown University, July 1997.

[17] “Program Slicing”, M.Weiser, IEEE Transactions on Software Engineering, V10, N4, pages:352-357.

[18] J.P.Tsai and S.H.Yang, “Monitoring and Debugging of Distributed Real-Time Systems”, 1-19, IEEE Computer Society Press. ISBN: 0471160075.