Journal of Computer and System Sciences 80 (2014) 953–972

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Software control flow error detection and correlation with system performance deviation ☆

Atef Shalan *, Mohammad Zulkernine

School of Computing, Queen's University, Kingston, Ontario, K7L 3N6, Canada

ARTICLE INFO

Article history:
Received 23 September 2012
Received in revised form 15 March 2013
Accepted 27 August 2013
Available online 11 February 2014

Keywords:
Control flow error
Error detection
Runtime monitoring
Component-based software
Performance analysis
Connection Dependence Graph (CDG)
Error state parameters
Regression analysis

ABSTRACT

Detecting runtime errors helps avoid the cost of failures and enables systems to perform corrective actions prior to failure occurrences. Control flow errors are major impairments of system dependability during component interactions. Existing control flow monitors are susceptible to false negatives due to possible inaccuracies of the underlying control flow representations. Moreover, avoiding performance overhead and program modifications are major challenges in these monitoring techniques. In this paper, we construct a connection-based signature approach for detecting errors among component interactions. We analyze the monitored system performance and examine the relationship of the captured error state parameters with the system performance deviation. Using the PostgreSQL 8.4.4 open-source database system with randomly injected errors, the experimental evaluation results show a decrease in false negatives using our approach relative to the existing techniques. It also demonstrates a significant ability of identifying the responsible components and error state patterns for system performance deviation.

1. Introduction

With the humongous expansion of the software roles in today's life, ensuring software quality is becoming more essential practice. Practical analysis of failure manifestation in a software system involves tracking the underlying causes (errors) and sources (faults). Detecting runtime errors is important to assure system resilience. It helps avoid the failure costs and enables systems to perform corrective actions prior to failure occurrences. Control flow errors are major impairments of system performance and other quality attributes during component[1] interactions. Operational environments of software systems are the main source of faults that can cause these errors [2]. Software faults with respect to operating systems and hardware faults with respect to computational devices are the main causes of control flow errors in software systems [3]. Recent industry trends of microprocessors aim to manufacture small size products with low supply voltage and high frequency. These trends are responsible for more susceptible transient hardware errors and control flow errors in software systems operating on these microprocessors [4].

☆ The earlier work of this research appeared in [1].

* Corresponding author.
E-mail addresses: atef@cs.queensu.ca (A. Shalan), mzulker@cs.queensu.ca (M. Zulkernine).

[1] *A software component* is a source code and data container that has identified boundaries and specified interfaces with other containers that together construct a software system.

http://dx.doi.org/10.1016/j.jcss.2014.02.004
0022-0000/

Error detection techniques mainly use signature-based approaches to monitor the control transitions among code instructions, system components, or basic blocks and detect their anomalies. These techniques detect errors by the use of watchdog, redundancy, assertion, or monitoring approaches. These approaches vary among each other by trading-off among four main goals: avoiding performance overhead, avoiding program modifications, avoiding the use of extra hardware, and increasing error coverage by detecting more errors of different types or among different scopes of control transitions. Signature-based control flow monitors can save more performance overhead than the other techniques and at the same time they do not require any additional hardware or software redundancy. Moreover, based on the signature structure, these techniques can also increase the error coverage. However, in these techniques, a unique signature is assigned to and injected into each block. Then, errors are detected by validating runtime signatures of the blocks against the program architecture. Some signature-based monitors use finite state automata [5] or branch traces [6] to represent a software architecture. However, the majority of these techniques rely on Block-based CFGs (BCFGs) for this purpose [7,4,8].

A BCFG is a CFG structure where the graph nodes are basic blocks[2] and edges are the control transitions among these blocks. However, the BCFG structure is remarkably complex and it lacks accuracy by allowing some illegal control branches in its representations. Due to the possible inaccuracies of control flow representations, the error detection techniques may encounter improper information about the internal system states and other consequent flaws in their results. Therefore, in these techniques, validating the control flow transitions may not be feasible for large software systems and may be susceptible to false negatives due to the inclusion of some illegal control branches in the BCFG. Moreover, maintaining simple control flow representation is important to practically implement these analysis techniques in large software systems. In addition, the existing techniques do not differentiate between connection invocations and control transitions based on the language constructs. As a result, these techniques are hard to use for backtracking program execution traces and validating their successful terminations. Moreover, the basic block partitioning in these techniques decreases the precision of error localization at the statement level inside these blocks.

System performance is an important quality attribute and of high concern with respect to several software systems (*e.g.*, production, control, and multimedia). Performance analysis techniques (*e.g.*, load test) can be time consuming. Expensive efforts are required to set up the test environment, prepare the test data, execute the test, and analyze the results [11–13]. In general, these techniques utilize selective performance log variables or execution state variables to reason about system performance. Control flow monitors can perceive details about system execution states that can be used to reason about system performance and other quality attributes. The Connection Dependence Graph (CDG) [14] can also enrich the error detection with useful error state parameters that correlate to system performance. Utilizing the distribution of errors and error state parameters among components in the performance analysis can help identify responsible components and error state patterns (specific combination of the error state parameter values) for performance deviations among system runs.

By extending our previous work [1], this paper utilizes a simple and accurate CFG representation based on the CDG to construct a connection-based signature approach for control flow error detection. We also correlate the detected error parameters to system performance deviation. We first describe our connection-based control flow signature structure in which, we partition the program components into code blocks. Each partition (or block) is associated with a CDG to represent the control structure of its code content. Next, we provide the control flow monitor structure and error checking algorithm based on these CDGs. We use the Multiple Correlation Coefficient (MCC) to examine the relationships of the captured error state parameters with the system performance deviation and identify the responsible components and error state patterns. The error detection approach is evaluated using PostgreSQL open-source database [15] and results of detecting control flow errors in different software versions with variable numbers of randomly altered code statements are rendered. The results show a decrease in false negatives using our approach relative to the existing signature-based techniques that use the BCFG representation. It also demonstrates the ability to identify highly correlated components and error state patterns to the system performance deviation with high "significance test"[3] values.

Fig. 1 presents an overview of the proposed technique in two steps: preparation and monitoring. In the preparation step, we instrument the software system in order to enable the generation of runtime signatures during its execution. A *connection-based control flow signature* is a runtime state describing the currently invoked connection during a system runtime. A program is partitioned into Connection Implementation Blocks (CIBs) (a code block). Each CIB is represented using a CDG. These CDGs are used to compute the runtime signatures. Signatures are injected in the program code to allow exporting the actual runtime transitions in a form of runtime signatures. In the monitoring step, we capture these runtime signatures and use them to verify the system control flow transitions. The monitor validates these runtime signatures by using the valid control flow transitions specified by the program CDGs and finally, an error report is generated. Using log data of a number of system runs, the errors and their parameters are correlated to the system performance at the component level and at the error state pattern level. The results of this process describe the correlation of each component and error state pattern with the system performance deviation.

The main contributions of this work include a control flow monitor using connection-based signatures. Our signature-based control flow monitor does not require any additional hardware or software redundancy. The underlying control flow

[2] A *basic block* is a maximal set of ordered code statements in which the execution starts at the first statement and ends at the last one. With the exception of the last statement, a block cannot include any control flow branching instruction [9,10].

[3] The significance test (F-test) checks if the resulting correlation coefficients have a Fisher distribution under the null hypothesis.

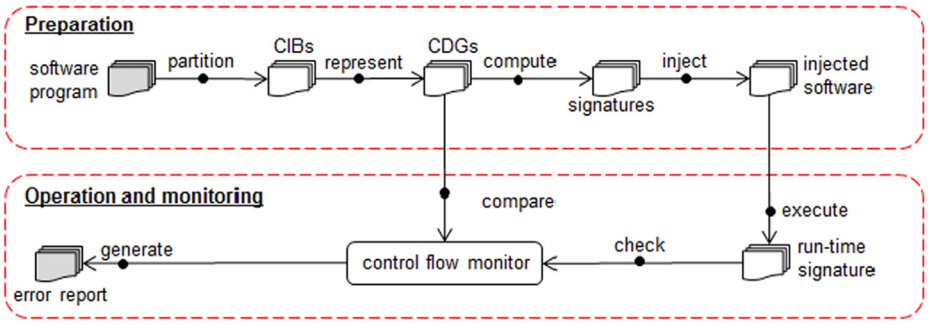


Fig. 1. Overview of the control flow error detection technique.

representation using CDG in the monitor decreases the possibility of false negatives due to its higher accuracy relative to the commonly used BCFG representation [14]. The connection-based signature structure allows complete coverage of all program connections and enables the tracking of their invocations and return addresses. Therefore, it can relate a runtime state to the execution traces of a software system in order to determine errors with respect to the unreturned connections (existing in the control stack) in case of improper program termination. The monitor helps locate the responsible component of each error to the code statement level. The use of the CDG in control flow error detection allows injecting only one line of code with respect to each graph node during program instrumentation. Given that, the CDG can significantly reduce the performance overhead of a control flow monitor in comparison to the existing techniques that inject two lines in each block. Our control flow error detection technique can be used to monitor the system behavior in computing environments where operational faults may cause control flow errors. Furthermore, a number of useful error state parameters characterizing the illegal control transitions are captured. Our regression analysis is able to identify the responsible components and error state patterns for performance deviation based on the captured error state parameters and thus, it avoids frustrating efforts of separate performance tests.

The remaining sections of this paper are organized as follows. Section 2 provides some background information and discusses the related work. Section 3 describes the proposed connection-based signature structure. Section 4 provides the control flow monitor structure and explains the control flow error checking algorithm. Section 5 provides regression analysis of system performance deviation dependency on the errors and error state parameters. Section 6 presents our experimental evaluation setup and process along with our findings. Finally, we present the conclusions and future work in Section 7.

2. Background and related work

Prior to proceeding in this paper, we state some basic terminology about software architecture and system reliability. Then, we discuss the related work of the proposed technique.

2.1. Background

In this section, we describe some background information about the BCFG and the CDG representations of software architectures.

2.1.1. Connection anatomy

A *connection* is defined as a mean by which a component interacts with another component (*e.g.*, procedure call, event trigger, and data access) [16]. A connection represents a control flow and/or data transfer channel between two architectural points: connection invoker and implementer. At the architectural level, a connection invoker represents an input port and a connection implementer represents an output port of a component interface. At the program code level, a connection invoker (or simply, a connector) and a connection implementer are specified based on the programming paradigm. The connectors involve procedure calls, events, and data accesses [17]. Unlike data access connectors, the procedure call and the event connectors allow synchronous and asynchronous control transfer, respectively. A *procedure call* connector models the flow of control between two components and allows data transfer between them through the use of formal parameters. An *event* is an instantaneous initiation or termination of a process that is handled by another or by the same software component.

A *connection implementer* expresses a procedure declaration statement, an event handler, or data object. For the former two connectors (procedure call and event trigger), a *Connection Implementation Block* (*CIB*) is a portion of program code that represents the body of the connection implementer and that is executed each time one of its connectors is visited. For example, in the procedural programming paradigm, a CIB is equivalent to a procedure body and a connector is a procedure call. In object oriented programming, a CIB is a class method including constructors and destructors and a connector is an implicit method call or an automatic invocation of a class constructor or destructor. In event-driven programming paradigm, a CIB is an event handler and a connector is an event trigger. A *connection invocation* is the process of control and/or data

transfer from the connection invoker to its CIB. A *connection return* is the process of control and/or data transfer from the CIB back to the connection invoker. Relative to a connector, we refer to the containing CIB as Invocation Code Block (ICB). Therefore, an ICB is a code block where a connection is invoked (the caller block) and a CIB is a code block where a connection is implemented (the callee block).

2.1.2. Block-based Control Flow Graph (BCFG)

In its general form, a control flow graph (CFG) is a representation of all possible execution paths that might be traversed through a program during its operation. The graph nodes represent the basic code blocks and the graph edges represent the control flow transitions among these blocks. In most presentations, there are two designated virtual nodes: the start node and the end node where the control enters and leaves the control flow graph, respectively. A *control flow sequence* is a number of connected nodes. We define a *control flow branch* as a special sequence of three nodes and two edges connecting them. An *illegal control flow branch* indicates two successive transitions where the first transition is correct (*i.e.*, complies with the software architecture) while the second one is incorrect.

The BCFG is a CFG structure that is intensively used in the existing control flow monitors. The BCFG partitions a software program into basic code blocks and represents them as graph nodes [18,4]. In these techniques, a basic block is defined as a maximal set of ordered code instructions in which the execution starts at the first and ends at the last statement. A block cannot include any control flow branching instruction with an exception only for the last statement [19]. Examples of control flow branching instruction are jump instructions, procedure calls, and control structure instructions [10]. The basic blocks can be determined at the level of the assembly code or at the high-level programming language code [20].

The partitioning of a program into basic blocks does not completely cover all the program connections. Some connection invocations may not be associated with any block. Very often, one or more procedure calls are invoked in a condition expression of a language control structure statement (construct). These calls are not associated with any block since they exist in the start of a compound statement that may have several other branches. Likewise, several connections may also exist in an expression statement. These connections are not associated with any blocks since in spite of being included in one statement, they cannot fit to a basic block. These ignored connections introduce CFG complexity through supplementary edges to represent their invocations. The following subsection illustrates the possible illegal control flow sequences that may result due to the additional complexity by an ignored connection with respect to the program partitioning.

2.1.3. Connection Dependence Graph (CDG)

Connection Dependence Graph (CDG) represents software control flow based on the dependencies among system connectors [14]. Unlike the BCFG, the CDG provides a simple representation of software control flow with lower probability of illegal control flow sequences. CDG outlines all the possible connector sequences that can be invoked during system execution. In its general form, a CDG is a directed connected cyclic graph $D = (N, E)$, where N is the set of nodes and E is the set of edges. Each node of the graph is a connection invocation (*e.g.*, a procedure call and event trigger). Each edge in the graph indicates a dependency relationship between two connections. A dependency relationship between two connectors specifies a deterministic invocation order due to a call sequence or an execution path based on control structure statements of a programming language.

The use of a CDG rather than a BCFG in error detection techniques allows complete coverage of all program connections. Unlike BCFG, the use of the CDG in these techniques can identify the error location at the statement level (where the connection exists) rather than the code block level. Furthermore, unlike BCFG, using CDG in control flow error detection allows injecting only one line of code with respect to each graph node. Given that, the CDG can significantly reduce the performance overhead of a control flow monitor in comparison to the existing techniques that inject at least two lines in each block.

2.2. Related work

Similar to error diagnosis, isolation, and recovery, error detection is an important aspect of fault tolerance technique to avoid system failures [21]. The expression “error detection” is sometimes used to indicate techniques for assuring system quality throughout the development phases including inspections, code reading, algorithm analysis, tracing, and control flow analysis [22]. However, these techniques belong to the fault avoidance and removal techniques for software reliability. Control flow analysis of these techniques aims to check the program control flow with respect to design problems (*e.g.,* unreachable code or poor program structures). Numerous types of errors are addressed by the existing error detection techniques. Examples of these errors are overflow, underflow, division by zero, memory allocation and deallocation, data flow, and control flow errors [23].

Control flow errors are major impairments to software system correctness. Operational environments of software systems are the main source of faults that can cause these errors [2]. Software faults with respect to the operating system [18] and hardware faults with respect to the address circuits, program counter, and memory elements are the main causes of control flow errors [3]. Most of the hardware causes of control flow errors are related to the computing devices. Between 33% and 77% of transient hardware faults lead to software control flow errors [2,8]. Recent industry trends in microprocessors aim to manufacture small size products with low supply voltage and high frequency. These trends are responsible for more susceptible transient hardware errors and control flow errors in software systems operating on these microprocessors [4].

Moreover, some special system environments may have further sources of control flow errors. For example, different space radiations may cause control flow errors in a software system operating in the space environment [10].

Several approaches were proposed to detect [18,2] and correct [24,4] runtime errors among the interactions of the architectural elements (*e.g.*, components, functions, object, process, task, state, etc.). Usually control flow error correction techniques use some form of redundancy to correct the runtime errors. The main goals of the current control flow error detection approaches are avoiding performance overhead [25,18,8], program modifications [7,18], and the use of extra hardware [6,26,19]. The goals also involve increasing error coverage by considering both control flow and data flow errors [27,4] or by considering intra-block and inter-block control flow transition errors [26,28,3]. Some of these goals contradict each other. As a result, most of the existing techniques aim to trade-off between the former two goals on one side and the latter two on the other side.

Control flow error detection techniques mainly use signature-based approaches to monitor the control transitions among code instructions, system components, or basic blocks and detect their anomalies. Existing techniques detect errors by the use of watchdog [26], redundancy [6,28,24], assertion [7,8], or monitoring approach [28,26,19]. Watchdog techniques aim to avoid the program modifications by using a watchdog processor. Michel et al. [25] describe an approach called WDP (Watchdog Direct Processing) which directly monitors the code instruction addresses of a main processor. By executing the same program operated by the main processor, the watchdog computes signatures of the executed instruction sequences and detects illegal execution paths. The major disadvantage of watchdog techniques is the need for additional hardware and the non-portability to various computing platforms [7].

Redundancy-based control flow error detection aims to detect and correct both control flow and data flow errors [29, 8]. Rebaudengo et al. [30] introduce data and code redundancy by performing a set of transformations on the high-level source code. Data flow errors are then detected by duplicating each variable and adding consistency checks after every read operation. Control flow errors are detected by duplicating the code instructions implementing each program operation and adding checks for verifying the consistency of the executed operations. Vemu et al. [2] present automatic correction of control flow errors using an algorithm involving addition of specific redundant code to the program. After assigning a unique Id for each program function, errors are detected by comparing the actual function Ids with the expected ones based on the program CFG. Errors are corrected by returning the control of the program back to the node that was being executed before the occurrence of a control flow error. Zarandi et al. [4] add redundant code instructions to a given program and detect the errors by comparing the actual control transitions and data transfers to the control and data flow graphs. Although these techniques go further beyond control flow error detection by addressing data flow errors and performing error correction, they involve a number of disadvantages. Redundancy is in general expensive and hard to obtain. By duplicating program variables and/or code instructions, the performance overhead will increase and system service will be degraded dramatically.

Assertion-based control flow error detection techniques aim to decrease the performance overhead and the use of extra hardware. They fortify a program with pre-inserted assertions in specific code locations. These assertions adjudge the control flow transitions among code instructions (or block) based on the injected signatures of each instruction (or block). Alkhalifa et al. [7] identify branch-free intervals (BFI) of code and insert assertions into the beginning and the end of these intervals. They detect the errors through modifications made on the GNU Compiler Collection (GCC). Vemu et al. [28] automatically embed extra code instructions into a program to continuously update run-time signatures and compare them against pre-assigned values. Although these techniques aim to decrease the performance overhead, the system performance is still degraded by the execution of the enormous number of assertions inserted into the program.

Signature-based control flow monitors aim to validate the control flow transitions in a separate software monitor. Thus, these techniques can save more performance overhead and at the same time they do not require any additional hardware or software redundancy. Signature-based monitoring approaches partition a software program into basic blocks [9,25,5,27, 28,26,2,19,3]. Some of these techniques use finite state automata [5] or branch traces [2] to represent software architecture. However, the majority of them use the CFG for the same purpose [9,27,28,26,19,3]. The graph nodes are basic blocks and the graph edges are the control transitions among these blocks.

In general, signature-based monitors assign a unique signature for each basic block and inject it at the beginning and the end of the block to allow runtime monitoring of the program control transitions at its start and exit points. Control flow errors are detected by validating runtime signatures of the blocks against the program CFG. Aiming to increase the error coverage, Nicolescu et al. [27] present an error detection technique by considering intra-block transitions. By replicating program operations, the intra-block control flow is validated by using a check-pass flag between the original and the replicated operations. The inter-block control flow is checked by using dedicated global control variables describing the current program state. These variables allow the technique to predict the next set of instructions and to compare them with the actual transitions. Inserting a check-pass flag after each line of code inside a block could exaggerate the performance mitigation and overhead. Maghsoudloo et al. [8] present an automatic control and data flow error detection and correction technique. Oh et al. [9] present a signature-based software monitoring technique for inter-block control flow. Sedaghat et al. [18] present a software-based error detection technique using encoded signatures. These techniques assign different arbitrary numbers or derive unique encoded signatures to the program basic blocks. Then, they follow the signature-based validation method mentioned earlier. The major limitation of these techniques is the use of basic block partitioning.

Due to complexity, some code statements neither fit into any basic block nor can they be partitioned into smaller blocks. Examples of these code complexities are control structures with procedure calls in their condition expressions and expression statements with multiple procedure calls. Connections that are invoked in these statements are ignored in the program

partitioning and thus, they introduce CFG complexity through supplementary edges to represent their invocations. Therefore, the ignored connections are becoming major factors in introducing illegal control branches in the BCFG representation. Consequently, in the existing techniques, validating the control flow transitions may be susceptible to false negatives due to the inclusion of these illegal control branches in the BCFG. In comparison to the BCFG, the CDG is a more accurate control flow representation that decreases the false negatives.

Few techniques have attempted to overcome the limitations of the BCFG representation in the control flow error detection. Wu et al. [24] aim to avoid the decrease in error coverage due to the fan-in (multiple inward edges) nodes where several graph nodes are succeeded by a single node. In their error detection approach, they modify the CFG by eliminating these fan-in nodes. Li et al. [10] aim to decrease the performance overhead by avoiding the variations in the program block sizes and their effects on the injected monitoring instructions. They regroup the small basic blocks into larger sizes to achieve almost equal sizes of all basic blocks. Then, they apply a unified checking method for the regrouped basic blocks represented using a modified CFG. Due to the simplicity of the CDG, its use in control flow monitoring can decrease the performance overhead while avoiding the aforementioned problem of the basic block partitioning.

3. Connection-based signature structure

In connection-based signature structure, a unique signature is given to each connection of a software program. We identify a connection by its Invocation Code Block (ICB), and its corresponding CDG node of this invocation block. In this section, we partition the program into Connection Implementation Blocks (CIBs). These CIBs are used as ICBs to describe the runtime signatures with respect to the connection invocations. For each CIB, a CDG is obtained from its code block to represent its control structure. We also describe our signature form and its injection method.

3.1. Program partitioning

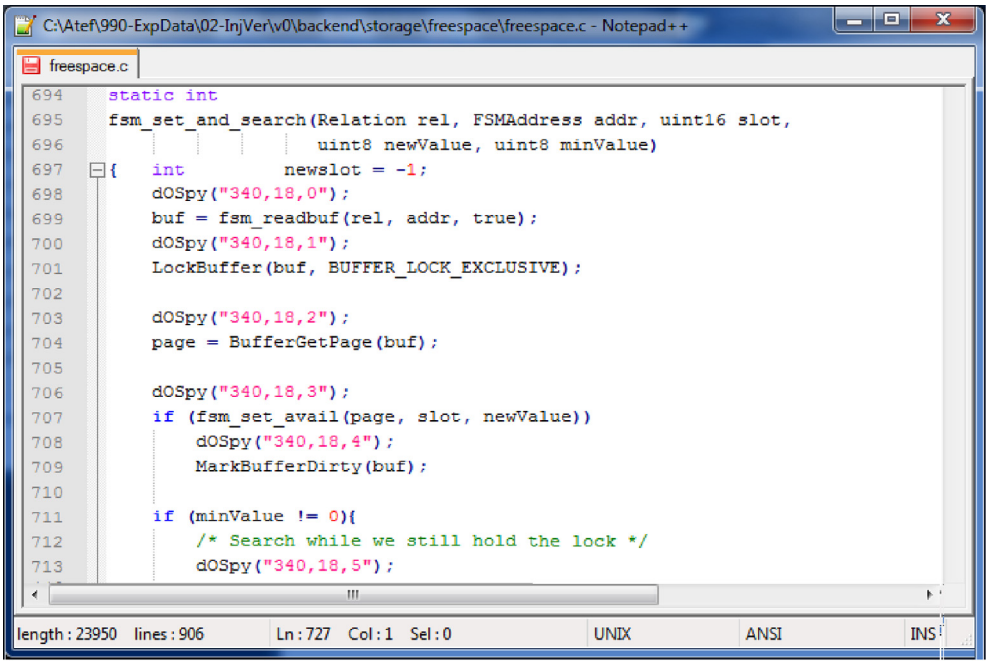
During a program runtime, the currently executing component and the current ICBs are important ingredients to identify a connection in our signature structure. To map each connection to its component and ICB, we first provide a unique Id for each program component. We then partition each component according to its CIBs. Some other code segments of the program may exist outside these CIBs. However, these code segments are usually preprocessing, prototyping, and variable declarations which do not involve any connection invocations. Consequently, they are not assigned any signatures in this technique.

We use these CIBs as program partitions and represent the code of each CIB using a separate CDG. Thus, a CDG will differentiate between two types of control transitions: connector invocations based on procedure calls (nodes) and subsequent control flow transitions based on programming language constructs (edges). This distinction is important to track the inter-component interactions and to create a call (or execution) stack. Both transition types take place at each CDG node. Relative to any CDG node, a connection invocation (node invocator) indicates the target CIB. A subsequent transition (node successor) indicates the next connection in the same CIB based on the node successors. However, a node invocator always precedes all subsequent transitions of a CDG node. When the target CIB of a node invocator returns, a transition through any successor of this node can take place. Using CDG, a system state is described by the invoked connector and a state transition indicates a transfer of the execution among these connectors.

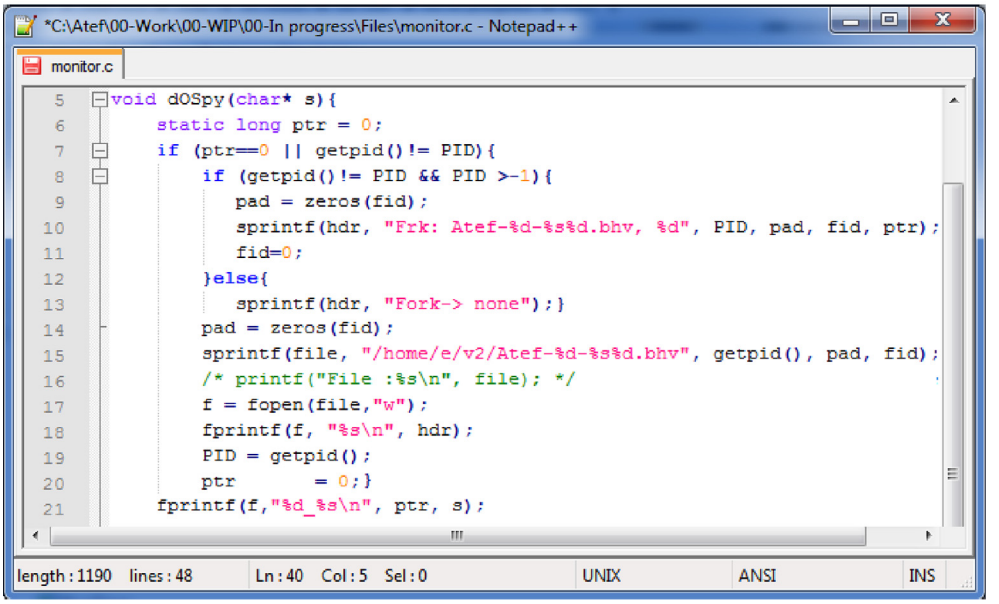
3.2. Signature form and injection method

A runtime signature is a unique string that describes system execution state with respect to its current control flow transition. A signature string identifies the current executing component, CIB, and connector. Consequently, a signature is formed as “Component Id, CIB Id, Connector Id”. For example, the signature “163, 9, 5” indicates that component 163 is executing the 9-th CIB and invoking the 5-th connector in this code block. If several connections exist in one code statement, a compound signature is used to describe the system state at the time of executing this code statement. An example compound signature is “163, 9, 5; 6; 7”. This signature indicates that the currently invoked connector exists in a statement that includes other connectors. The invocation order of these connectors is: “5, 6, 7”. Each signature is injected in the program code according to the invoked connector statement location. We inject a line of code before each statement that includes one or more connectors. The injected code line writes the included signature to an output file stream.

Fig. 2(a) displays a sample file (*/backend/storage/freespace/freespace.c*) of the PostgreSQL database server software project. The figure shows injected signatures in this file by using our in-house developed reverse architecting toolkit [31]. The toolkit injects a signature as a call parameter to a procedure *dOSpy*(). The tool automatically writes the implementation code of procedure in a file (called *monitor.c*) and declares it in a header file (called *monitor.h*). These two files are added to the PostgreSQL makefile and compiled with the software project. A call to the procedure *dOSpy*() exports the runtime signatures to an output stream in order to be captured by the control flow monitor (see Fig. 4). The code inside the procedure *dOSpy*() can be found in Fig. 2(b). Fig. 2(b) shows the implementation of the procedure *dOSpy*(). This procedure exports the runtime signatures to the output stream and manages the output data based on the operating system forking. Managing the output data based on the operating system forking is essential to control the log counters of each concurrent process and maintain appropriate sequences of log records. The procedure *dOSpy*() is called at each injected line of code by the control flow monitor. The call parameter at each line is the control flow signature of the corresponding connector ID.



(a) Automatically injected signatures in the PostgreSQL database software.



(b) The procedure *dOSpy()* that exports the runtime signatures to the output stream.

Fig. 2. Signature form and injection.

4. Control flow monitor structure

The error detection technique uses a CDG to represent a program architecture. A runtime monitor uses this CDG to validate the control flow transitions among system connectors. In this section, we describe the proposed monitor structure as shown in Fig. 3. The instrumented software systems generate runtime signatures representing the actual control flow transitions. The monitor receives these signatures and validates them using the stack manager and the error detector. The stack manager maintains a control stack for the connection invocations during system runtime. The error detector validates runtime signatures by comparing them to the CDG by using the control stack. If errors are found, the monitor generates an error report. The use of control stack allows detecting not only illegal branches but also unreturned connections in case

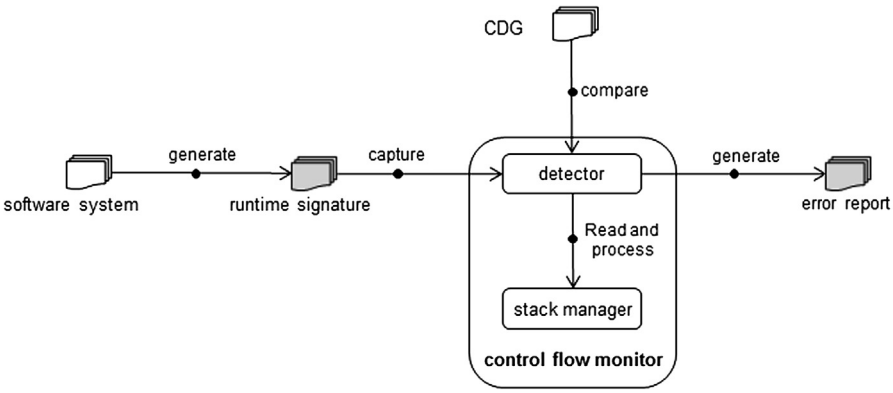


Fig. 3. The control flow monitor structure.

Algorithm 1 Control flow error detector.

Input: current state s, runtime signature t.
Output: error report (erroneous state and parameters).

```
01. if (system is running) do
02.   if (t == system.exit()) do
03.     stack.clear()
04.   else
05.     find current CDG using s.component and s.CIB
06.     find CDG node n_i using s.connector
07.     if (t == n_i.invocator) do
08.       stack.push s;
09.     else
10.       n_j = stack.peek()
11.       if (t ∈ n_j.successors) do
12.         stack.pop()
13.       else
14.         generate error report of illegal transition;
15.       end if
16.     end if
17.     s = t
18.   end if
19. else
20.   for (i = 1 to stack.size()) do
21.     generate error report of unreturned transition;
22.   end for
23. end if
```

of an improper system termination. In the following subsections, we explain the details of the control stack and the error detection algorithm. Then, we describe the detected error state parameters.

4.1. Stack manager

The control stack tracks the connection invocations of a system during its runtime to allow validating connection return transitions. In case of an improper system termination, this stack helps detect incomplete execution traces and unreturned connections. The control stack includes a list of signatures describing previously invoked but not yet returned connectors. Each stack element indicates a connection invocation that took place inside the CIB of the preceding stack element. A connector in the stack element at the i-th position can return only if all the connectors in the higher stack positions return beforehand.

4.2. Control flow error detector

Control flow error detector runs each time a runtime transition is observed by the monitor. The detector relates the current system state to the program CDG and then compares the runtime signature to the possible valid transitions based on this CDG. The detector interacts with the stack manager by adding or removing elements. In case proper termination using a *system.exit*() operation is observed, the detector triggers emptying the stack to avoid reporting its remaining elements as unreturned transitions.

Algorithm 1 presents the control flow error detection method. The algorithm checks control flow for illegal transitions (Lines 1–18) while the system is running and for unreturned transitions (Lines 19–23) after the system exits. Line 1 checks

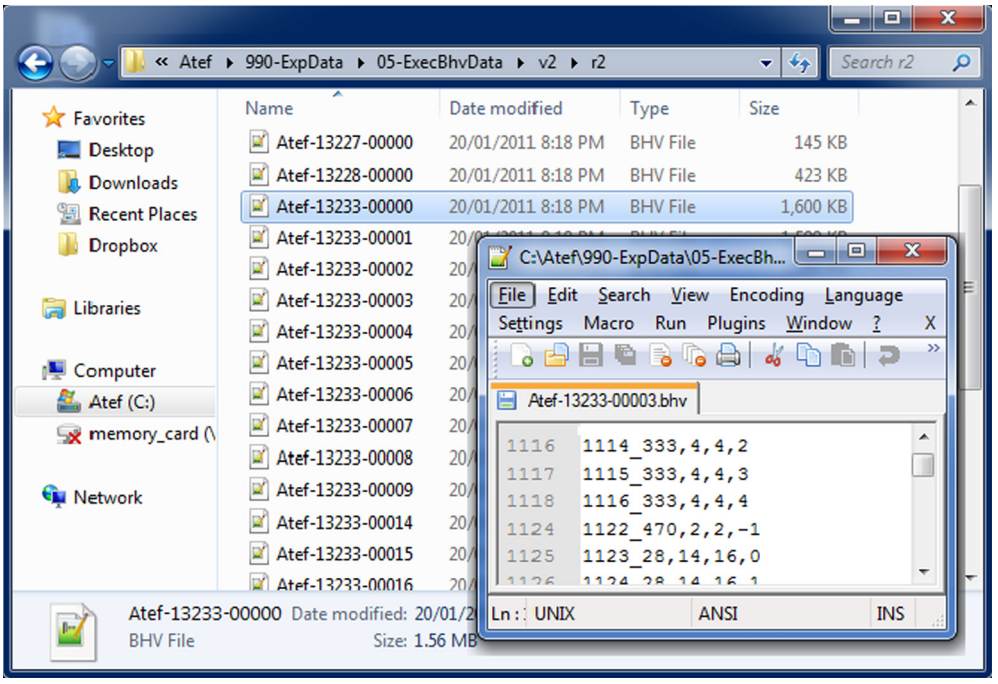


Fig. 4. The exported runtime signatures by an executed version of the PostgreSQL database.

the system running state. If the system is running, Lines 2–4 determine if the current transition is *system.exit*() and consequently, clear the stack if it is true. Lines 5–16 validate the actual transition t based on the current system state s. First, Line 5 determines the CIB currently executed from the system state s using the component and CIB IDs of the state s. Line 6 determines the current node in the CDG of the currently executed CIB using the connector ID of the state s. Lines 7–16 validate the transition t against the invocator edge of the current CDG node n_i. If the transition is validated, then s is pushed into the stack for future control return checking. If it is not validated against the node invocator, Lines 10–15 validate the transition t using the control stack. Line 10 peeks the top node of the control stack. Line 11 validates the transition t against the successor transitions of the top stack node. If it is validated using these successors, the stack top is popped out. Otherwise, an error report is generated. In Line 7, the transition t becomes the current state s. Finally, Lines 19–23 generate an error report for the unreturned transitions that are remaining in the control stack.

4.3. Error state parameters

Two types of control flow anomalies are detected in our work: illegal branches and unreturned connections. As defined in Section 2, an illegal branch indicates a control transfer that does not comply with the software architecture represented using the CDGs. Unreturned transitions indicate an improper program termination. A program terminates appropriately, if the control stack is empty or if it exits using an "end" operation (*e.g.*, "*system.exit*()" library function in C-language"). In case of an improper exit of a program, the remaining stack elements are considered as control flow errors and classified as unreturned connections.

The other error parameters determined in this work include error characteristics: external, repeated, and stacked. An external transition is a control transfer to a CIB in a component that is different from the current component. A repeated transition is a control transition that has been classified as valid in an earlier system state with different stack elements. A stacked transition indicates that the target runtime signature of the incorrect transition already exists in an element that is not on the top of the stack. Error location is another parameter that is determined using our technique. This parameter is directly derived from the runtime signature. Using the error location, one can determine the most erroneous or most intact system components.

The determined error parameters provide important comprehension about error states. For example, unreturned connections may indicate silent failures. Illegal branches can be used to predict other failure types (*e.g.*, value or performance failures) depending on the values of the other parameters: external, repeated, and stacked. The determined error parameters provide important comprehension about a system control flow error state. These error parameters can also be used to determine the components that are highly correlated with performance delay.

Fig. 4 shows some exported runtime signatures by the executed software versions. The signatures are organized in files that are named after the concurrent process Id and file counters. In each file, a list of runtime signatures represents the

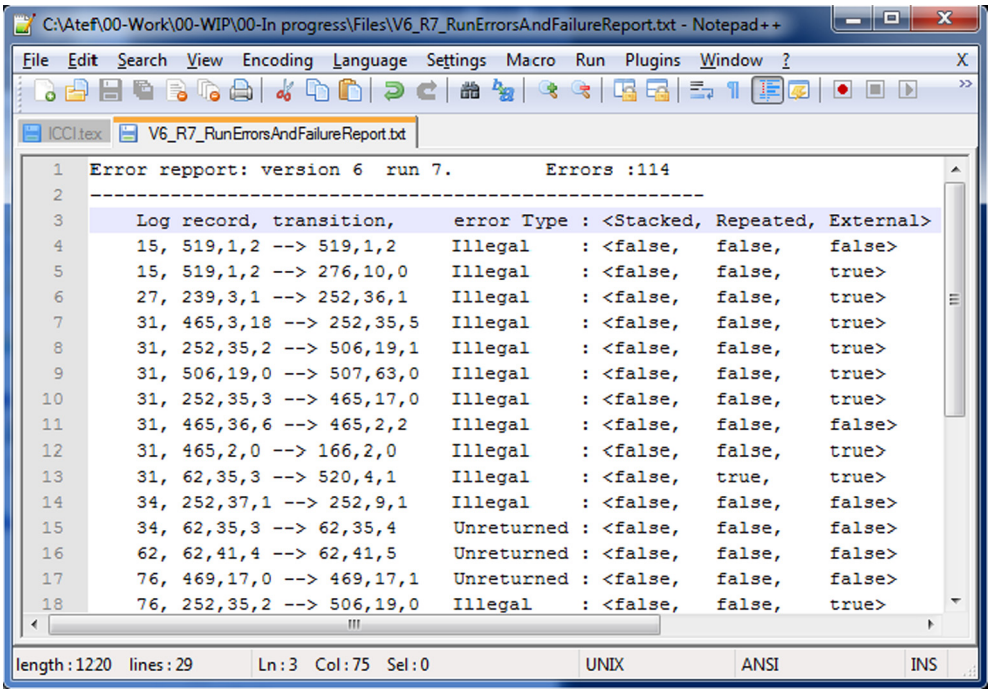


Fig. 5. The error results of a sample system run of the PostgreSQL database.

control flow path during an execution interval of this process. Each line in this file indicates a control flow transition. Fig. 5 displays sample error log records generated by the control flow monitor.

5. Performance analysis using error state parameters

During system execution, errors may occur and affect system performance and other quality attributes. In this section, we study the dependency relationship between the error state parameters and system performance. We examine the effect of individual components and error state patterns on the whole system performance aiming to identify the components and the error state patterns responsible for performance deviation. An *error state pattern* is a vector of specific error state parameter values.

5.1. Analysis methodology

The relationship between error state parameters and system performance depends on the performance measure used in the regression analysis and the interpretation of the error parameters as input variables to the regression function. The performance measure used in our analysis describes the system performance deviation for early/late service delivery. First, this performance deviation can be measured for any system run by subtracting the specified runtime from the actual runtime of this run. Second, our aim in this analysis is to identify components and error state patterns responsible for performance deviation. For this purpose, we craft and utilize three statistical interpretations (error measures) of the captured error state parameters of each run. These error measures express the following.

- The total number of errors in each component i: α_i.
- The error distribution among error state patterns in each component i: β_i.
- The number of errors of each error state pattern j in the whole system: γ_j.

The dependency relationships between each of these error measures and the system performance are indicated using the correlation coefficients between them and system performance deviation. The resulting correlation coefficients can identify the components and error state patterns responsible for the performance deviation by selecting the highest among them. They also allow comparing the dependency relationships of the different error measures with the system performance.

A correlation coefficient describes the degree of dependency between an independent and a dependent variable. Several correlation methods are used intensively in the current literature to determine the dependency between two variables (*e.g.*, Pearson product-moment and Spearman's correlation coefficients). In our analysis, some error measures of the error state parameters include only a single variable. These error measures are the total numbers of errors of each component and total numbers of errors of each state pattern. For each run r, each of α_i and γ_j defines a single variable associated

with component i and error state pattern j, respectively. The other error measure (β_i) defines a vector of j variables (β_i) associated with each component i. To correlate these multiple error measures with the performance deviation, we use a more generalized correlation method: Multiple Correlation Coefficient (MCC), H. Abdi [32]. MCC generalizes the standard coefficients of correlation and assesses the quality of the prediction of multiple independent variables with a dependent variable. In the following subsection, we describe the regression analysis of our approach.

5.2. Multiple correlation coefficient

A multiple correlation coefficient $R^2_{Y.XJ}$ correlates multiple independent variables $x_1, x_2, \ldots, x_J$ and an observed (dependent) variable y and provides an estimate of the combined influence of the independent variables on the dependent variable. In case of orthogonal independent variables, $R^2_{Y.XJ}$ is equal to the sum of the squared coefficients of correlation between each independent variable and the dependent variable. To generalize our approach for any combination of error parameters, we do not assume this orthogonality. Therefore, we derive $R^2_{Y.XJ}$ as the squared correlation between the predicted and the actual values of the performance deviations. The predicted performance deviations are calculated as variations in the dependent variable that are directly related with the regression on the error measures (the regression sum of squares). Thus, the regression function is described by the matrix b of $J+1 \times 1$ coefficients of the independent variables $x_1, x_2, \ldots, x_J$ as derived in Eq. (1) as follows.

$$b = (\tilde{X}X)^{-1}\tilde{X}Y \tag{1}$$

Notice that $\hat{Y} = Xb$ where $\hat{Y}$ indicates the predicted dependence variable from the independent variables $1, x_1, x_2, \ldots, x_J$. Therefore, $R^2_{Y.XJ}$ is expressed as the regression sum of squares $SS_{regression}$ divided by the total sum of squares ($SS_{regression} + SS_{error}$) in Eq. (2) as follows.

$$R^2_{Y.XJ} = \frac{\tilde{b}\tilde{X}Y - \frac{1}{N}(\tilde{1}Y)^2}{\tilde{Y}Y - \frac{1}{N}(\tilde{1}Y)^2} \tag{2}$$

where X is an $N \times (J+1)$ matrix of the values of $1, x_1, x_2, \ldots, x_J$ in N rows and Y is an $N \times 1$ vector of the corresponding observations for the dependent variable. In this equation, $\tilde{b}$, $\tilde{X}$, and $\tilde{Y}$ are the matrix transposes of b, X, and Y, respectively, and $\tilde{1}$ is a matrix of 1's that is conformable with the transpose of Y. In order to assess the significance of the resulting correlation coefficient, we compute the F-test as follows.

$$F = \frac{R^2_{Y.XJ}}{1 - R^2_{Y.XJ}} \times \frac{N - J - 1}{J} \tag{3}$$

This significance test provides a measure (F ratio) for the probability that $\frac{R^2_{Y.XJ}}{1-R^2_{Y.XJ}}$ and $\frac{J}{N-J-1}$ have the same variance. The F ratio is distributed under the null hypothesis as a Fisher distribution. If the value of F is greater than 1, then the test is significant and we can reject the null hypothesis.

5.3. Data preparation

We designate and describe the inputs of our regression analysis using MCC. Precisely, we describe the independent and the dependent variables of this correlation method. Let us represent a system S as a number of I components as $\{c_i\colon\ i = 1, 2, \ldots, I\}$. A system is operated R times and in each run r, a number of errors E occurs in the system. Each error $v_e\colon\ e = 1, 2, \ldots, E$ is described by a number of error state parameters $\xi_1, \xi_2, \ldots, \xi_K$, where K is the number of error state parameters. For each run r, the performance deviation is observed and reported as a number δ_r in microseconds. This performance measure is calculated as the actual runtime minus the execution time specified in the desired Service Level Agreement (SLA). We define a number of error state patterns $\mu_1, \mu_2, \mu_3, \ldots, \mu^J$, where each pattern μ_j expresses a vector of values of the provided error state parameters, *i.e.*, $\mu_j = \langle\tilde{\xi}_1, \tilde{\xi}_2, \ldots, \tilde{\xi}_K\rangle$, where $\tilde{\xi}_k$ is an occurrence value of the error state parameter ξ_k. These *occurrence values* of the error parameter are selected from the actual error occurrences in system runs. Thus, an error state pattern describes an error state that occurred at least once during the system runs. Hence, the number of error state patterns J describes all the actual occurrences of error state parameter combinations during the system runs. Table 1 lists example patterns determined using the error state parameters described in Section 4. These patterns are used by the regression analysis of our experimental evaluation. The table lists the error state parameters in column 1 and the patterns in columns 2 to 9. The rows represent the possible truth combination of the three error parameters. Thus, using these error state parameters, there are possibly 2^3 state patterns that could characterize an error state.

Using this data, we create three measures to express the total number of errors (α_i) and the number of errors of each error state pattern j (β_i) in each component i, and the number of errors (γ_j) of each error state pattern j in the whole system. We use these measures to study and compare their relationships with the system performance deviation during the

Table 1
The possible error patterns determined using the illegal transition characteristics in Section 4.

	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7	μ_8
External transition	0	0	0	0	1	1	1	1
Repeated transition	0	0	1	1	0	0	1	1
Stacked transition	0	1	0	1	0	1	0	1

reported R runs. Using these measures, we create three corresponding matrices X_i^{α}, X_i^{β}, and X_j^{γ}, where X_i^{α} and X_i^{β} are determined for each component c_i and an X_j^{γ} is determined for each pattern μ_j in the whole system.

$$X_i^{\alpha} = \begin{bmatrix} x_0^{\nu,i} \\ x_1^{\nu,i} \\ \vdots \\ x_r^{\nu,i} \\ \vdots \\ x_R^{\nu,i} \end{bmatrix}, \quad X_i^{\beta} = \begin{bmatrix} x_{0,0}^{\mu,i} & x_{1,0}^{\mu,i} & \cdots & x_{j,0}^{\mu,i} & \cdots & x_{J,0}^{\mu,i} \\ x_{0,1}^{\mu,i} & x_{1,1}^{\mu,i} & \cdots & x_{j,1}^{\mu,i} & \cdots & x_{J,1}^{\mu,i} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{0,r}^{\mu,i} & x_{1,r}^{\mu,i} & \cdots & x_{j,r}^{\mu,i} & \cdots & x_{J,r}^{\mu,i} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{0,R}^{\mu,i} & x_{1,R}^{\mu,i} & \cdots & x_{j,R}^{\mu,i} & \cdots & x_{J,R}^{\mu,i} \end{bmatrix}, \quad X_j^{\gamma} = \begin{bmatrix} x_{j,0}^{\mu} \\ x_{j,1}^{\mu} \\ \vdots \\ x_{j,r}^{\mu} \\ \vdots \\ x_{j,R}^{\mu} \end{bmatrix}, \quad \text{and} \quad Y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_r \\ \vdots \\ y_R \end{bmatrix} \tag{4}$$

Eq. (4) lists the matrices of the error measures and the performance measure used in the regression analysis. In this equation, $x_r^{\nu,i}$ is the number of errors in component i during run r. $x_{j,r}^{\mu,i}$ is the number of errors of state pattern j in component i during run r. $x_{j,r}^{\mu}$ is the number of error states of pattern j during run r in the whole system. Let ρ_i^{α}, ρ_i^{β}, and ρ_j^{γ} be the correlation coefficients of the matrices X_i^{α}, X_i^{β}, and X_j^{γ} with the matrix Y, respectively. Thus, both ρ_i^{α} and ρ_i^{β} describe the dependency relationship between each component and system performance deviation. However, in this dependency relationship, ρ_i^{α} considers only the number of errors without any details about the error state parameters and ρ_i^{β} considers these parameters by involving the number of errors of each state pattern. Finally, ρ_j^{γ} describes how effective an error state pattern μ_j on the system performance deviation.

6. Experimental evaluation

We evaluate the error detection technique using the PostgreSQL 8.4.4 database software system [15]. We aim to examine the efficacy of our approach to detect control flow errors in a number versions which are altered randomly to produce different errors during runtime. For this purpose, 10 different program versions were generated by randomly altering some code statements and altering the program control flow structure accordingly, without introducing compilation errors. Each version is operated and monitored using 10 different experimental data sets. In the following subsections, we describe the experimental environment, process, and setup. Then, the latter subsections present the results.

6.1. Experimental environment

PostgreSQL is an industrial open source object-relational database system. It is used by hundreds of customers all over the world in several business fields including government, education, finance, retail, and telecom systems. The source code of this application exceeds 900,000 lines of code and is written in C-language. In our experiment, we run PostgreSQL on Ubuntu 11.04 operating system. The source code is analyzed and results are computed on Windows 7 operating system using Java 2 Platform Enterprize Edition (J2EE). The main challenge of this experiment is to obtain detailed architectural information about the software in use. We overcome this problem by using our reverse architecting toolkit library [31]. This library includes a number of tools that can obtain architectural information of C-based application programs. It also performs code injection according to the obtained architecture. The following subsections describe the details of the experimental setup and process.

6.2. Experimental process and setup

Prior to generating signatures and injecting them among other logging code, we use our reverse architecting toolkit [31] to compute the CDGs of the program CIBs. A unique signature is given to each connector based on the component Id and the block Id that includes it. We inject a line of code to export the current execution state as a runtime signature. A similar line of code is injected prior to each connector in the program (see the example in Fig. 2(a)). The injected code lines simply call a procedure implemented in a C-file that have been added and associated with a GNU Make file of the PostgreSQL project (see the example in Fig. 2(b)). This procedure manages the logging information in separate transition repository files and assures that the number of logs in each file does not exceed 100,000 transitions. It also arranges the transition repository

files according to the process Ids generated by the forking (concurrent execution) mechanism of the PostgreSQL database (see the example in Fig. 4). The database system is executed and the control flow transitions are validated according to the algorithm in Fig. 1. Depending on the current stack of each transition, error parameters are determined for each error. The detected errors are exported in log records and stored in a file repository. In the following paragraphs, we describe our methods for generating different software versions and for automatic software operation.

6.2.1. Software version generation

Ten versions of PostgreSQL database with different numbers of altered code statements are created using our toolkit [31]. Statements were altered in random components and random code locations. Two methods were applied to alter code statements randomly: repetition and removal. The repetition method duplicates a code statement and the removal method deletes a statement. The altered statements are selected randomly with some constraints to avoid compilation errors and to control the impact of the altered statements on the control flow. The chosen constraints assure that only expression statements are altered. They also avoid altering a statement, if it is the only one in its block and avoid repeating jump statements. The number of altered statements in these versions are chosen as $0, 3, 6, \ldots, 27$ for versions $0, 1, 2, \ldots, 9$, respectively. This incremental sequence of altered statement numbers is chosen to check their variable impact on the control flow errors of these versions. Note that, version 0 has 0 altered code statements, therefore, this version is considered to be the original version and is used to validate the outputs of the other versions with respect to the same inputs.

6.2.2. Automatic software operation and experimental data sets

Experimental data sets are chosen to allow maximum coverage of the database engine operations and to allow automatic execution of the different versions against these data sets. Ten different experimental data sets are created to cover the server, database, client, data, and tool management of the database engine. These data sets are created in the form of SQL files and executed as parameters to the database engine in the command lines of the shell programs (see Fig. 7). By including different lists of database commands in these SQL files, the provided experiments vary with respect to their performance loads. The experimental data set (E_9) executes the system for significantly longer time. The different performance loads allow comparing system execution times. The large experimental data set (E_9) aims to increase the executed code coverage during system runs.

An automatic operation toolkit was developed to allow configuring, compiling, and building the executable files of the different software versions (see Fig. 6). The toolkit also cleans previous runs' information and prepares for receiving new transition repository files. After performing these setup procedures, the toolkit operates each version individually against all the experimental data sets and arranges the transition repository files in a folder structure. The tool includes a number of shell programs operating on Red Hat and Ubuntu Linux systems. The execution start and end times are exported for each run in a performance log file. The execution time of each database version in each run is calculated in microseconds from the start and end times and compared to the runtime of version 0 with respect to the same experimental data sets. This subsection presents some samples of the shell scripts and SQL files used for automatic execution of the PostgreSQL database versions and experimental data sets.

Fig. 6 displays part of a shell script file for automatically compiling and operating the different PostgreSQL database versions and managing their outputs in a folder structure. This script is called through other shell programs that perform higher level tasks of the automatic execution including folder cleaning, switching between experimental data sets, and some other tasks. Fig. 7 presents a sample SQL file used as experimental data set during the automatic execution of the PostgreSQL database server versions. The SQL file performs some database server commands, creates tables, inserts data into them, and executes queries on these data.

6.3. Resulting errors in versions and runs

Fig. 8 shows the control flow errors detected in the different versions against the experimental data sets. Figs. 8(a)–8(j) display 10 runs (0 to 9), where run $0, 1, 2, \ldots,$ and 9 exploit the data sets $0, 1, 2, \ldots,$ and 9, respectively. In these figures, the X-axis represents the versions 0 to 9 and the Y-axis represents the total number of errors. The result of each run is displayed separately to ensure the visibility of the graphs since each run has a different vertical scale range. Fig. 8(k) displays the results from all the runs to allow comparing them.

In general, these graphs show the increase in the number of errors with the increase in the number of altered statements from version 0 to version 9. Version 0 is not mutated and thus, it represents the original version. Versions 1–5 included approximately 0, 131, 144, 122, and 163 control flow errors, respectively on average among all the runs. Versions 6–10 included relatively higher average of control flow errors in most versions. These versions approximately included 1072, 228, 94, 2512, and 100 control flow errors, respectively on average among all the runs. In these versions, we notice that versions 6 and 9 have relatively higher control flow errors than the other versions and versions 8 and 10 have lower control flow errors than the others. The reason behind the increase of control flow errors in versions 6 and 9 is the occurrence of errors in the first run (run 0). This run mainly initiates the database server (Postgres). The errors in this particular operation are always carried out across all the remaining experimental data sets. The errors in the server startup mean that there are some server processes that failed to initialize. Consequently, these initialization failures will generate more failures in any further use of these processes. The low number of errors in versions 8 and 10 is due to the randomness of the statement

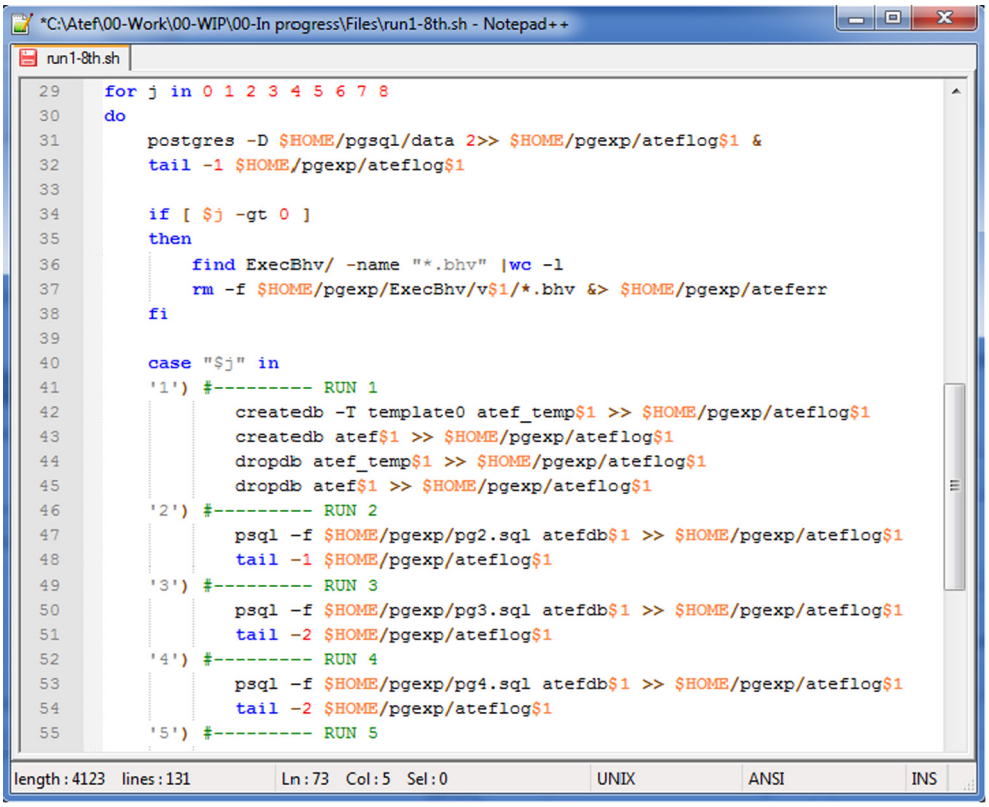


Fig. 6. Part of the shell script of the automatic operation toolkit.

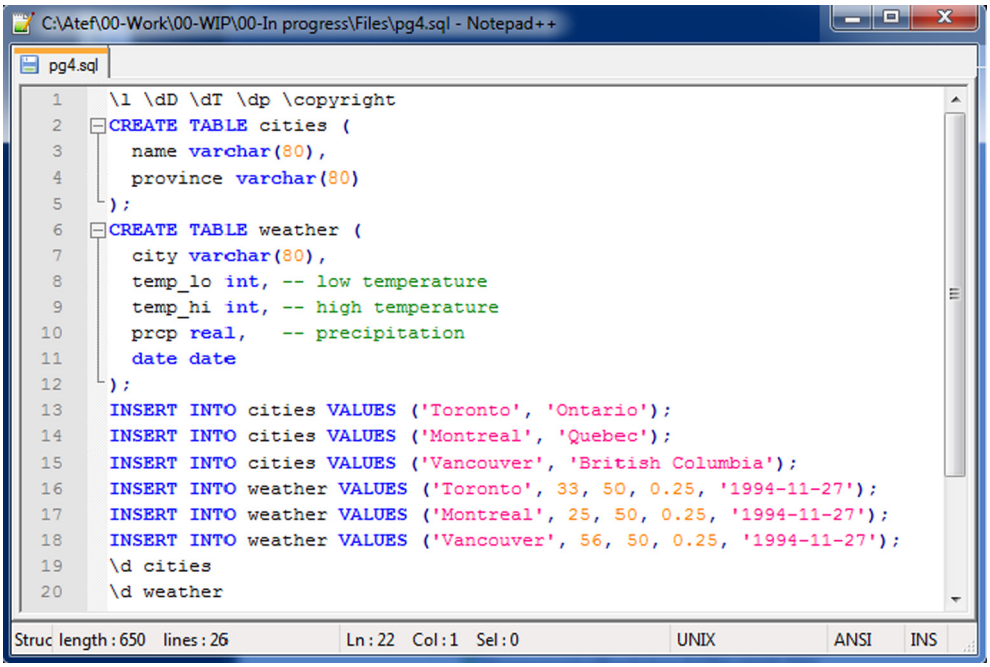


Fig. 7. A sample SQL file that is passed as a parameter to the automatic execution toolkit.

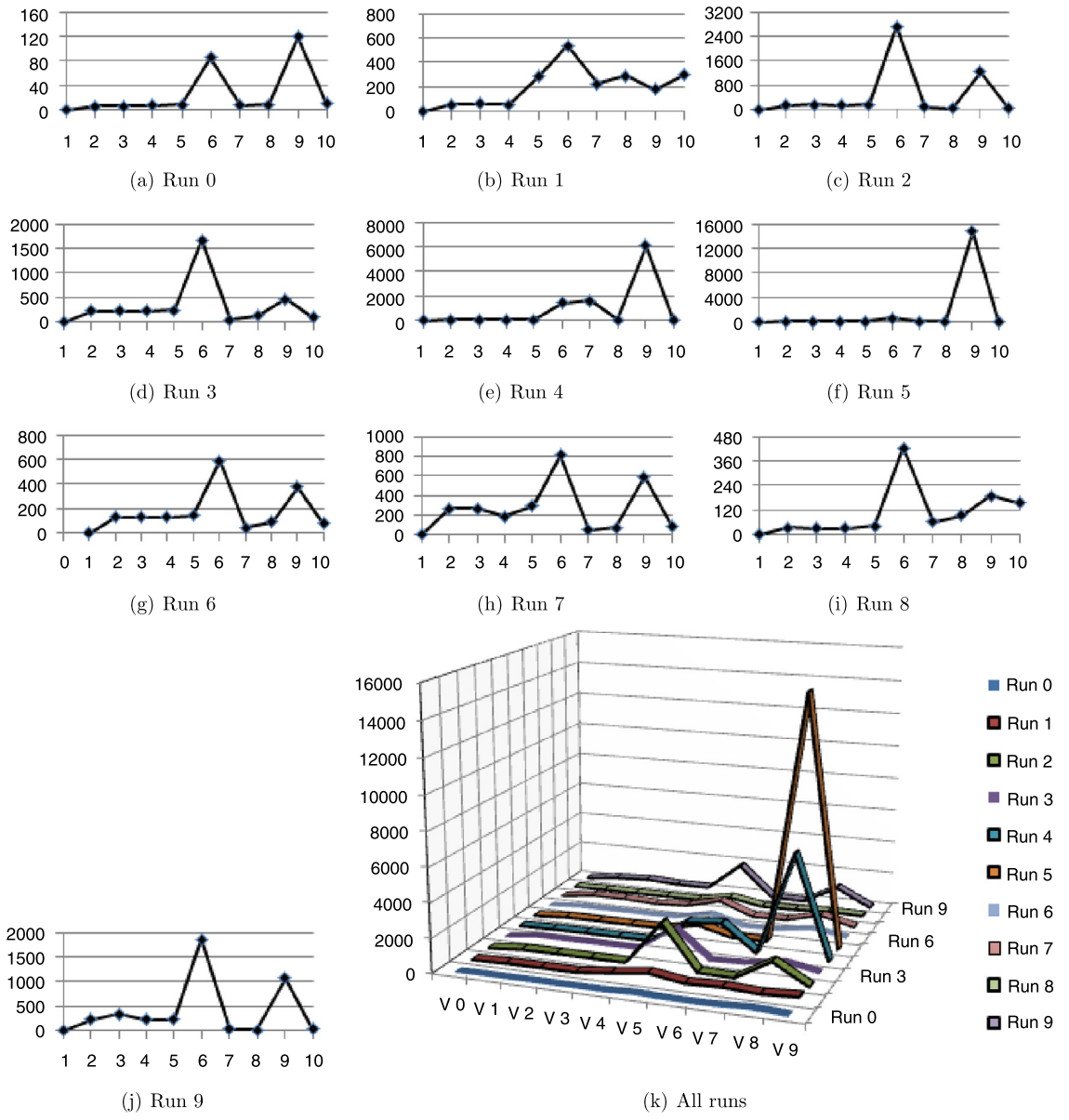


Fig. 8. The control flow errors in versions and runs.

altering method. It is noticeable that the altered statements in these versions are not affecting the control flow as much as other versions. This may be due to altering statements in unused or slightly used components with respect to the selected experimental data sets. It may also be due to the importance of the selected statements or to the statement altering method within the selected components.

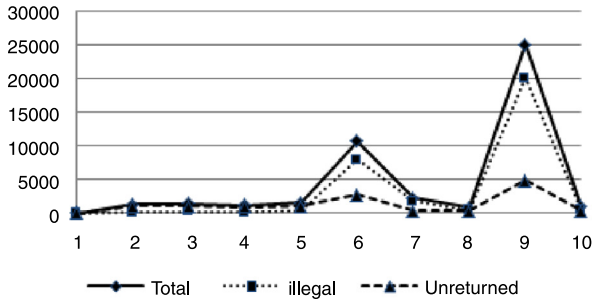
6.4. Resulting performance measures and error parameters

We present the actual execution times of the system runs with respect to the experimental evaluation of our control flow monitor in Table 2. The execution time of the system runs (columns) against the experimental data sets (rows) are approximated and displayed in seconds. These execution times are used to capture the performance deviation of system runs later in this section. From the table, most of the data sets operate the system from 1 to 25 seconds.

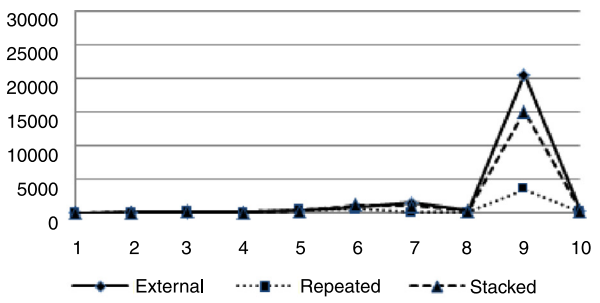
Fig. 9 presents the number of control flow errors according to the error types, characteristics, and highest occurrences in system components. The error types (illegal, unreturned) and characteristics (external, repeated, and stacked) are shown in Figs. 9(a) and 9(b), respectively. The X-axis represents the versions 0 to 9 and the Y-axis represents the total number of control flow errors. It is clear that most of the detected control flow errors are illegal branches for connections to other

Table 2
The actual execution times of the systems runs against the experimental data sets.

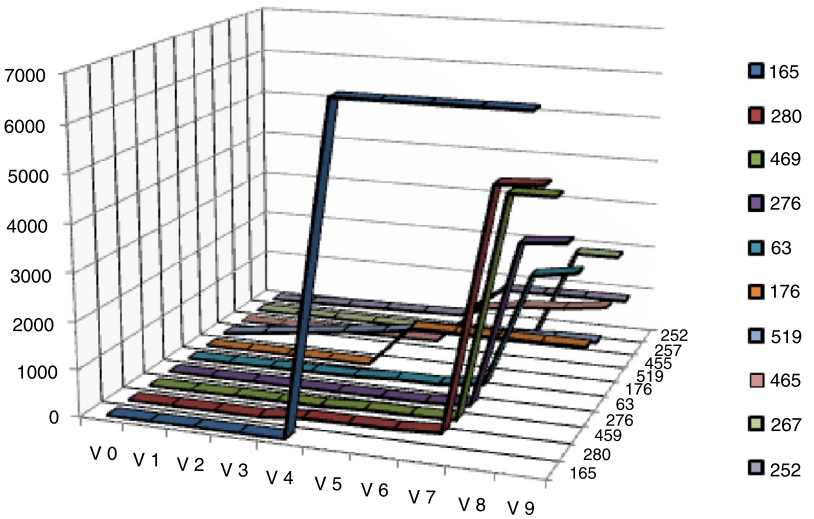
	run 0	run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8	run 9
E_0	1	1	1	1	1	1	1	1	1	1
E_1	23	23	23	23	23	25	8	12	23	5
E_2	12	12	12	12	13	18	3	3	6	1
E_3	10	10	11	11	10	11	4	5	5	1
E_4	20	19	18	20	20	23	7	3	17	1
E_5	7	7	7	6	7	7	5	5	991	2
E_6	7	6	5	5	5	6	3	5	4	1
E_7	5	5	6	4	6	6	4	3	5	1
E_8	9	9	9	8	8	9	7	8	9	2
E_9	1463	1250	1275	1341	1327	717	721	3	22,749	14,399



(a) Error types and total



(b) Error characteristics

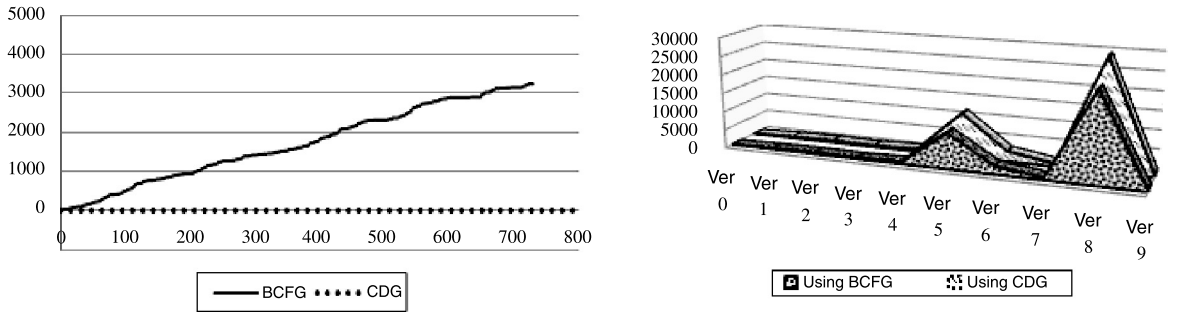


(c) System components with the highest numbers of errors

Fig. 9. The control flow error types, characteristics, and highest occurrences in system components.

system components (external transitions). The number of unreturned connections is relatively low and most of the detected errors are related to transitions that target components which do not exist in the current stack. Most erroneous transitions are not repeated, *i.e.*, they attempt to transfer the system to states that have not occurred previously in the current runtime.

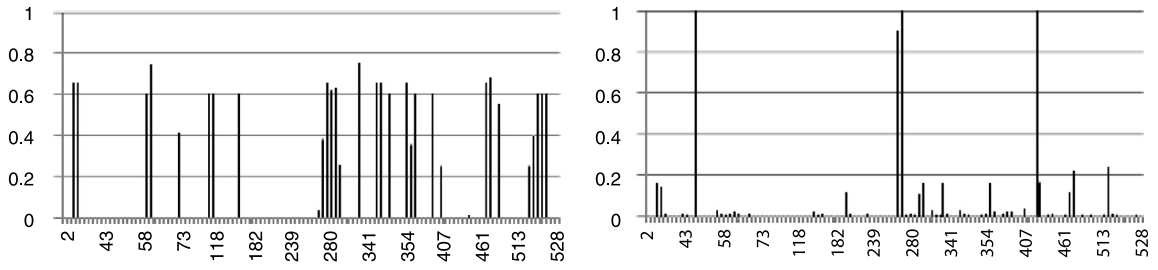
Fig. 9(c) presents the control flow errors according to the error location. The top 10 components with the highest control flow error occurrences are shown in the figure based on their component Ids. Components 165, 280, 469, 276, 63, 176, 519, 465, 267, and 252 correspond to dllist.c, pgstat.c, dynahash.c, autovacuum.c, xlog.c, pqsignal.c, aset.c, elog.c, sysv_sema.c, and scan.c, respectively. However, this high number of errors only indicates that the altered statements in the different versions are mainly distributed over these components. It also indicates that these components are mostly used by the experimental data sets during the system operation. The figure shows that our technique is able to locate the responsible component of each control flow error. Our technique can further narrow down the location to the code statement at which an illegal branch or unreturned connection has taken place.



(a) Ignored connections in different code sizes

(b) Detected control flow errors

Fig. 10. Comparing signature-based control flow error detection using CDG vs. BCFG.



(a) Version 7

(b) Version 8

Fig. 11. Example performance correlation with component errors of different software versions.

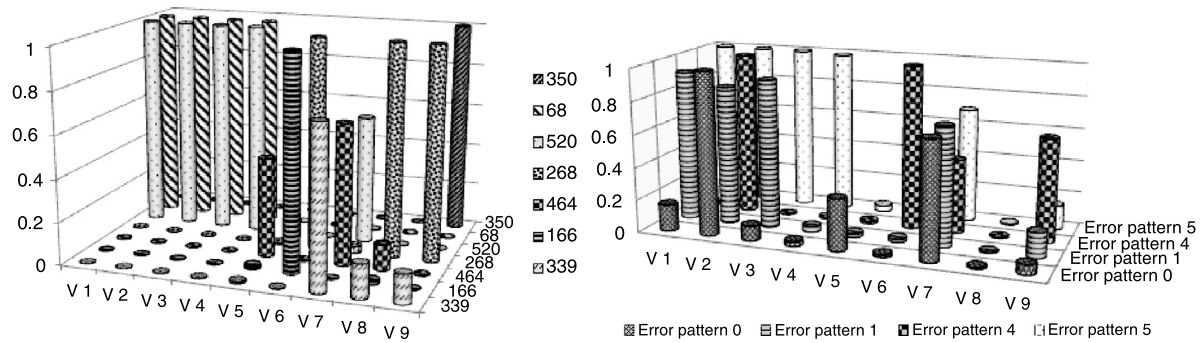
6.5. Comparing the usages of CDG and BCFG in control flow monitors

In Fig. 10, we present a comparison between our signature-based control flow error detection using CDG vs. the existing methods that use the BCFG representation. Fig. 10(a) presents the relationships between the number of ignored connections and the program code size in both CDG and BCFG. The results in this figure are obtained by virtually partitioning the PostgreSQL 8.4.4 software into a number of overlapped program segments with different sizes. For the purpose of this experiment, we specify these segments such that they have incremental sizes that vary in a range from one component to the whole system. Therefore, segments $1, 2, 3, \ldots, n$ include $1, 2, 3, \ldots, n$ components, respectively and the number of segments is equal to that of components. For each program segment, we derive the corresponding CDG and the BCFG using our CFG derivation mechanism [14]. We count the number of ignored connections in these graphs by parsing the program using our reverse engineering tool [31] and checking the existence of each procedure call in the two graphs. It is clear that, the CDG does not ignore any connection which is trivial due to its connection-based control flow graph structure. We also notice that the BCFG does not provide complete connection coverage and that, the ignored connections in this graph increase with the increase of the code size. This incomplete connection coverage is the main source of false negatives in the control flow error validations using the BCFG as shown in Fig. 10(b). The figure presents the total numbers of detected control errors in all system runs for different software versions with variable numbers of altered code statements. To derive this figure, each control transition is validated twice using the CDG and the BCFG of the PostgreSQL software. In this figure, the number of detected errors is larger with the validation using the CDG than that using the BCFG. Thus, undetected errors using the BCFG are successfully avoided in our technique.

6.6. Errors correlation with performance deviation

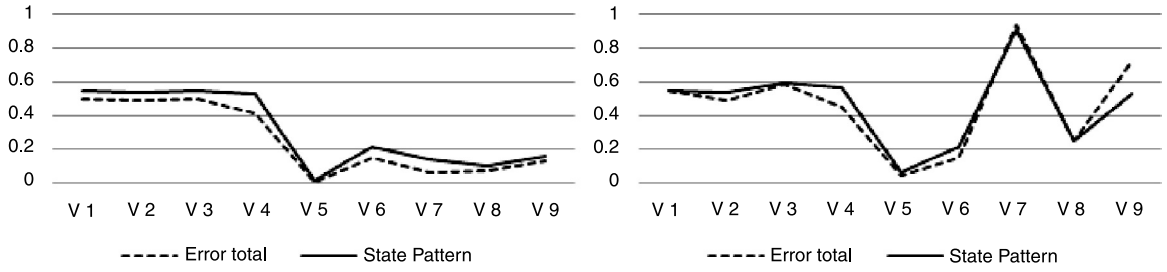
During the runs of the different software versions, some components included errors with different amounts and different error state patterns. Although the system has 730 components, only 42 components appeared to have errors during the runs of the system versions. The other components were either not executed for the used experimental data sets or they had no errors. Based on the randomly altered code statements in each component of the different versions, the component error behavior can be more or less effective on the whole system performance. Fig. 11 shows a number of components of two example versions on the X-axis and the error correlation on the Y-axis. The two patterns of error correlations of these versions contrast the distribution of performance deviation dependency on system components. In version 7, most of the components are equally correlated with performance deviation and in version 8 only few components are responsible for the deviation while the rest are not considerably effective.

We show the components and error state patterns of highest error correlation with system performance in the different versions in Figs. 12(a) and 12(b), respectively. In most versions, around 2 components out of the 730 are mostly responsible



(a) Components (b) Error state patterns

Fig. 12. The highest correlated components and error state patterns with performance deviation in the different versions.



(a) high correlation (b) high significance

Fig. 13. Comparing the high correlation and significance of the different versions based on error numbers and error state parameter correlations with performance deviation.

for the performance deviation. In the first 4 versions, the main responsible components are 68 and 520 which correspond to the "backend/catalog/aclchk.c" and "backend/utils/mmgr/mcxt.c". In the last 6 versions, component 268 correspondent to "backend/port/sysv_shmem.c" is the most common responsible component for performance deviation. Fig. 12(b) presents the highest correlated error state parameters with performance deviation in the different versions. Out of the 8 error state patterns, only four appear as error states during our experiments (p_0, p_1, p_4 and p_5). According to the relationships of these patterns with performance deviation, these patterns vary with respect to their error correlations with performance deviation in the different versions. For example, in version 2, all these patterns have high correlation coefficients while in version 8 all of them have low coefficients.

6.7. Correlation coefficients and significance (F-test)

Fig. 13 compares the high correlation (13(a)) and high significance (13(b)) of the different versions based on the two error measures: the total number of errors in each component i (α_i) and the number of errors of each error state pattern in each component i (β_i). In the two figures, the dashed line is the correlation using the number of errors and the solid line is the correlation of the error state parameters. The figures show that the correlation using the error state parameters (the β_i error measure) is in general slightly higher in some versions. It also shows that the significance of the correlation is almost identical for both error measures in all versions. However, with different and more detailed error parameters, we expect that the significance of correlation may increase with respect to the relationship of the error state parameters and performance deviation.

7. Conclusions

Control flow errors are major impairments to software system correctness. Several runtime monitoring approaches were proposed to detect control flow errors by using watchdog, redundancy, assertion, or monitoring approaches. The major limitations in these techniques involve high performance overhead, intensive program modifications, and use of extra hardware. Signature-based control flow monitors can save more performance overhead than the other techniques and at the same time, they do not require any additional hardware or software redundancy. However, existing signature-based control flow monitoring approaches rely on the BCFG to represent the system control flow structure. Therefore, in these techniques, validating the control flow transitions may not be feasible for large software systems and may be susceptible to false negatives due to the inclusion of some illegal control flow branches in the BCFG. The utilization of the CDG in our monitor decreases

the possibility of false negatives due to its higher accuracy relative to the BCFG. Moreover, our connection-based signature structure allows complete coverage of all program connections and it can determine the control flow errors with respect to the unreturned connections in case of improper program termination.

We present a runtime monitoring approach for control flow error detection using connection-based signatures. We also correlate the detected error state parameters to system performance deviation. Our connection-based signatures are constructed using a CDG representation. We provide our control flow monitor structure and checking algorithm based on the CDGs of the program partitions. The Multiple Correlation Coefficient (MCC) is used to examine the relationships of the captured error state parameters with the system performance deviation. An experimental evaluation of our approach is provided using PostgreSQL open-source database. The results show that our technique is capable of detecting control flow errors in 10 different versions with variable incremental numbers of randomly altered program statements with respect to the software control flow structures. The detected control flow errors in these versions are found to be mostly increasing with the number of altered program statements. Our control flow monitor is able to detect the improper system termination by distinguishing the unreturned connection invocations. Regarding the error state parameters, most of the detected control flow errors are due to illegal branches of external transitions and they are also not repeated in previous system states.

In comparison to the existing techniques that use BCFG representations, our approach is able to decrease the false negatives by validating control transitions using the CDG. Our approach detects the locations of the illegal transitions specified at the code statement level. It also detects the system success/failure termination state. A number of error state parameters characterizing the illegal control transitions are captured. The experimental results also demonstrate the ability to identify highly correlated components and error state patterns to the system performance deviation with high significance test values. Our regression analysis uses the captured error state parameters as inputs to the regression function and thus it avoids frustrating efforts of separate performance tests. Furthermore, unlike existing techniques, the usage of the CDG allows injecting only one line of code with respect to each graph node. Given that, our control flow monitor saves half of the performance overhead in comparison to the existing techniques that inject two lines in each block. Our connection-based signature structure helps locate the responsible component of each control flow error. It further narrows down the location to the code statement at which an illegal branch or unreturned connection has taken place.

Although our control flow monitor requires less code instrumentation than the existing techniques, we believe that, exporting a runtime signature prior to each call statement may still cause considerable overhead. We aim to alter our code instrumentation method to control the process of exporting and validating the system runtime signatures with minimal time delays. Our future work involves exploiting these parameters for diagnosing and repairing faults. Other future work can be classifying components using the correlation of the detected errors with the system performance. This classification helps determine plausible component crashes according to the history of its errors. We also aim to study the relationships (if any) between data flow errors and control flow error transitions. This relationship can later be used to construct a hybrid control and data flow approach with minimal amount of redundancy.

Acknowledgment

This research work is partially funded by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

[1] A. Mohamed, M. Zulkernine, A connection-based signature approach for control flow error detection, in: Proc. of the IEEE 9-th International Conference on Dependable, Autonomic and Secure Computing (DASC'11), December 2011, pp. 129–136.
[2] R. Vemu, S. Gurumurthy, J. Abraham, ACCE: Automatic correction of control-flow errors, in: Proc. of the International Test Conference (ITC'07), October 2007, pp. 1–10.
[3] J. Li, Q. Tan, J. Xu, Reconstructing control flow graph for control flow checking, in: Proc. of the International Conference on Progress in Informatics and Computing (PIC'10), vol. 1, December 2010, pp. 527–531.
[4] H. Zarandi, M. Maghsoudloo, N. Khoshavi, Two efficient software techniques to detect and correct control-flow errors, in: Proc. of the 16-th Pacific Rim International Symposium on Dependable Computing (PRDC'10), December 2010, pp. 141–148.
[5] C. Rabejac, J. Blanquart, J. Queille, Executable assertions and timed traces for on-line software error detection, in: Proc. of the 26-th Annual International Symposium on Fault-Tolerant Computing (FTCS'96), IEEE Computer Society, 1996, pp. 138–147.
[6] M. Fazeli, R. Farivar, S. Miremadi, A software-based concurrent error detection technique for powerpc processor-based embedded systems, in: Proc. of the 20-th International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05), IEEE Computer Society, 2005, pp. 266–274.
[7] Z. Alkhalifa, V. Nair, N. Krishnamurthy, J. Abraham, Design and evaluation of system-level checks for online control flow error detection, IEEE Trans. Parallel Distrib. Syst. 10 (6) (June 1999) 627–641.
[8] M. Maghsoudloo, N. Khoshavi, H. Zarandi, CCDA: Correcting control-flow and data errors automatically, in: Proc. of the CSI International Symposium on Computer Architecture and Digital Systems (CADS'10), September 2010, pp. 99–104.
[9] N. Oh, P. Shirvani, E. McCluskey, Control-flow checking by software signatures, IEEE Trans. Reliab. 51 (1) (March 2002) 111–122.
[10] A. Li, B. Hong, Online control flow error detection using relationship signatures among basic blocks, Comput. Electr. Eng. 36 (January 2010) 132–141.
[11] H. Malik, B. Adams, A. Hassan, Pinpointing the subsystems responsible for the performance deviations in a load test, in: Proc. of the IEEE 21-st International Symposium on Software Reliability Engineering (ISSRE'10), November 2010, pp. 201–210.
[12] J. Krizanic, A. Grguric, M. Mosmondor, P. Lazarevski, Load testing and performance monitoring tools in use with ajax based web applications, in: Proc. of the 33-rd International Convention on Information Communication Technology (MIPRO'10), May 2010, pp. 428–434.
[13] R. Mansharamani, A. Khanapurkar, B. Mathew, R. Subramanyan, Performance testing: Far from steady state, in: Proc. of the IEEE 34-th Annual Computer Software and Applications Conference Workshops (COMPSACW'10), July 2010, pp. 341–346.
[14] A. Mohamed, M. Zulkernine, A control flow representation for component-based software reliability analysis, in: Proc. of the International Conference on Software Security and Reliability (SERE'12), January 2012, pp. 1–10.

[15] P.G.D. Group, Postgresql featured users, Website, 1991, http://www.postgresql.org/about/users, last visited: April 1991.
[16] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison–Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
[17] N. Mehta, N. Medvidovic, S. Phadke, Towards a taxonomy of software connectors, in: Proc. of the 22-nd International Conference on Software Engineering (ICSE'00), ACM, 2000, pp. 178–187.
[18] Y. Sedaghat, S. Miremadi, M. Fazeli, A software-based error detection technique using encoded signatures, in: Proc. of the 21-st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'06), IEEE Computer Society, 2006, pp. 389–400.
[19] Y. Wu, G. Gu, K. Wang, An improved CFCSS control flow checking algorithm, in: Proc. of the International Workshop on Anti-counterfeiting, Security, Identification (IWASID'07), April 2007, pp. 284–287.
[20] R. Iyer, R. Iyer, Z. Kalbarczyk, Z. Kalbarczyk, Hardware and software error detection, Website, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.7571, last visited: June 2012.
[21] L. Pullum, Fault Tolerance Techniques and Implementation, Artech House, ISBN 1-58053-470-8, 2001.
[22] W. Peng, D. Wallace, Software error analysis, Website, Gaithersburg, MD, 2012, http://hissa.nist.gov/HHRFdata/Artifacts/ITLdoc/209/error.htm, last visited: June 2012.
[23] G. Luecke, J. Coyle, J. Hoekstra, M. Kraeva, Y. Li, O. Taborskaia, Y. Wang, A survey of systems for detecting serial runtime errors: Research articles, Concurr. Comput. 18 (15) (December 2006) 1885–1907.
[24] Y. Wu, G. Gu, S. Huang, J. Ni, Control flow checking algorithm using soft-based intra-/inter-block assigned-signature, in: Proc. of the International Multi-Symposiums on Computer and Computational Sciences (IMSCCS'07), August 2007, pp. 412–415.
[25] T. Michel, R. Leveugle, G. Saucier, A new approach to control flow checking without program modification, in: Proc. of the 21-st International Symposium on Fault-Tolerant Computing (FTCS'91), June 1991, pp. 334–341.
[26] E. Borin, C. Wang, Y. Wu, G. Araujo, Software-based transparent and comprehensive control-flow error detection, in: Proc. of the 4-th International Symposium on Code Generation and Optimization (CGO'06), IEEE Computer Society, 2006, pp. 333–345.
[27] B. Nicolescu, Y. Savaria, R. Velazco, SIED: Software implemented error detection, in: Proc. of the 18-th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03), IEEE Computer Society, 2003, pp. 589–597.
[28] R. Vemu, A. Jacob, CEDA: Control-flow error detection through assertions, in: Proc. of the 12-th IEEE International Symposium on On-Line Testing (IOLT'06), IEEE Computer Society, 2006, pp. 151–158.
[29] A. Meixner, D. Sorin, Error detection using dynamic dataflow verification, in: Proc. of the International Conference on Parallel Architecture and Compilation Techniques (PACT'07), September 2007, pp. 104–118.
[30] M.M. Rebaudengo, M.S. Reorda, M. Torchiano, M. Violante, Soft-error detection through software fault-tolerance techniques, in: Proc. of the 12-th International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'99), 1999, pp. 210–218.
[31] A. Mohamed, M. Zulkernine, Software reverse architecting toolkit library for c-based programs, Website, http://research.cs.queensu.ca/~atef/RArch.html, last visited: June 2012.
[32] H. Abdi, Multiple correlation coefficient, in: Encyclopedia of Measurement and Statistics, SAGE, Thousand Oaks, CA, USA, 2007.