Dynamic Analysis for Diagnosing Integration Faults

Leonardo Mariani, *Member*, *IEEE*, Fabrizio Pastore, *Member*, *IEEE*, and Mauro Pezzè, *Senior Member*, *IEEE*

Abstract—Many software components are provided with incomplete specifications and little access to the source code. Reusing such *gray-box* components can result in integration faults that can be difficult to diagnose and locate. In this paper, we present *Behavior Capture and Test (BCT)*, a technique that uses dynamic analysis to automatically identify the causes of failures and locate the related faults. BCT augments dynamic analysis techniques with model-based monitoring. In this way, BCT identifies a structured set of interactions and data values that are likely related to failures (failure causes), and indicates the components and the operations that are likely responsible for failures (fault locations). BCT advances scientific knowledge in several ways. It combines classic dynamic analysis with incremental finite state generation techniques to produce dynamic models that capture complementary aspects of component interactions. It uses an effective technique to filter false positives to reduce the effort of the analysis of the produced data. It defines a strategy to extract information about likely causes of failures by automatically ranking and relating the detected anomalies so that developers can focus their attention on the faults. The effectiveness of BCT depends on the quality of the dynamic models extracted from the program. BCT is particularly effective when the test cases sample the execution space well. In this paper, we present a set of case studies that illustrate the adequacy of BCT to analyze both regression testing failures and rare field failures. The results show that BCT automatically filters out most of the false alarms and provides useful information to understand the causes of failures in 69 percent of the case studies.

Index Terms—Dynamic Analysis, diagnosis, fault localization, false positive filters, regression failure analysis, field failure analysis.

1 Introduction

Software systems often include components, hereafter *gray-box* components, that are available with poor specifications and only give a partial view of the internal details. For example, many vendors offer Off-The-Shelf (OTS) components that generate complex reports from various data sources [1], [2], and several websites offer plugins that extend application capabilities [3]. Many of these OTS components and plugins are offered without source code and with informal (and incomplete) specifications.

Lack of access to source code and incomplete specifications harms the integration of *gray-box* components and can lead to integration faults with unpredictable effects. For instance, several investigations of aerospace problems indicate integration faults and incomplete specifications as major sources of critical software failures in aerospace missions [4], [5]. Static analysis techniques can identify and remove some faults, but require access to the source code or need formal specifications, and generate many false positives (FP) that reduce their practical applicability [6], [7], [8].

Limited availability of source code and specifications complicates debugging as well [9]. State-of-the-art debugging techniques propose different heuristics to identify fault locations by comparing the code executed in failed and successful executions [10], [11], [12], [13], [14], [15]. These techniques are effective, but require access to source code.

Dynamic analysis produces information that can help developers identify and localize faults, even in the presence of gray-box components. When analyzing gray-box components, dynamic analysis techniques monitor system executions by observing interactions at the component interface level, derive models of the expected behavior from the observed events, and mark the model violations, hereafter *anomalies*, as symptoms of faults [16], [17].

State-of-the-art dynamic analysis techniques produce useful information, but suffer from limitations that hinder their effectiveness and reduce their practical applicability:

- Most techniques focus on a single type of interaction, ignoring other ways of interacting that can be important in the component behavior. For example, Raz et al. focus on the data involved in the communication among components, but do not consider the interaction order [18], while Wasylkowski et al. focus on the interaction order, but do not cope with the data involved in the communication [17].
- Most techniques produce many false positives that are difficult to eliminate and time-consuming to inspect [18].

- *L. Mariani is with the Department of Informatics, Systems and Communication, University of Milano Bicocca, Edificio U14, viale Sarca 336, 20126 Milano, Italy. E-mail: mariani@disco.unimib.it.*
- *F. Pastore is with the Faculty of Informatics, University of Lugano, via Giuseppe Buffi 13, 6900 Lugano, Switzerland. E-mail: fabrizio.pastore@usi.ch.*
- *M. Pezzè is with the Faculty of Informatics, University of Lugano, via Giuseppe Buffi 13, 6900 Lugano, Switzerland, and the Department of Informatics, Systems and Communication, University of Milano Bicocca, Edificio U14, viale Sarca 336, 20126 Milano, Italy. E-mail: mauro.pezze@usi.ch.*

Manuscript received 30 July 2009; revised 26 Mar. 2010; accepted 1 Oct. 2010; published online 19 Oct. 2010.
Recommended for acceptance by S.C. Cheung.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2009-07-0192.
Digital Object Identifier no. 10.1109/TSE.2010.93.

- The currently available techniques cannot relate multiple model violations that depend on the same fault, thus providing developers with information hard to filter and interpret.

This paper proposes a solution, called *Behavior Capture and Test (BCT)*, to analyze failures and diagnose faults. BCT is based on incremental dynamic analysis techniques that extract useful information without expensive storage consumption, relates multiple model violations, and reduces the impact of false positives. This paper advances our previous work [19], [20] and the state of the art in automatic fault analysis by doing the following:

- Combining dynamic analysis and model-based monitoring to describe possible failure causes and fault locations. BCT provides information about failure causes by sizing a structured set of interactions and data values likely related to failures, and provides information about fault locations by indicating components and operations that are likely responsible for failures.
- Combining classic dynamic analysis techniques (Daikon) with incremental finite state generation techniques (kBehavior) to produce two kinds of dynamic models that capture complementary aspects of component interactions: *I/O models* capture information about the data exchanged between components, while *interaction models* capture the method invocations generated by a component when interacting with other components.
- Extracting information about likely causes of failures by automatically relating the detected anomalies.
- Filtering false positives in two steps. In the first step, BCT identifies and eliminates false positives by comparing failing and successful executions with heuristics already experienced in other contexts [21], [22]. In the second step, BCT ranks the remaining anomalies according to their mutual correlation and uses this information to push the related likely false positives far from the top anomalies.
- Presenting a set of case studies that indicate the effectiveness of the proposed technique.

The effectiveness of BCT depends on the quality of the dynamic models extracted from the program. The models are influenced by the samples of the input space that are used to derive the models themselves. BCT works particularly well when analyzing failures that occur after extensively sampling the successful execution space. The data provided by the case studies reported in this paper show that BCT is indeed effective in at least two important cases:

- Diagnosing faults that cause regression failures, that is, faults introduced by fixing or extending the program functionality. This happens quite frequently when systems, like Eclipse, are extended with plugin available in many versions [23].
- Diagnosing faults that cause field failures. This happens quite frequently when failures are due either to integration problems or to rare and faulty event sequences that passed unit testing [24], [25], [26], [27].

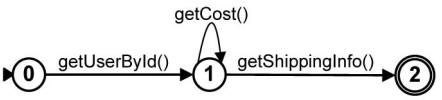


Fig. 1. The FSA associated with method `order(Cart cart)`.

This paper is organized as follows: Section 2 overviews the *BCT* solution. Section 3 presents model generation techniques. Section 4 defines a technique to reduce the number of false positives and to discover related anomalies. Section 5 formally describes the strategy to heuristically relate anomalies. Section 6 shows the results obtained by applying BCT to several case studies. Section 7 discusses the related work. Section 8 summarizes the main contributions of our research.

2 Behavior Capture and Test

BCT is a technique that builds models of component interactions from successful executions of software systems and uses these models to analyze failures and diagnose faults.[1] Successful executions are executions that satisfy the oracles or the user expectation, while failing executions are executions that are either aborted before normal termination (*program crashes*) or that violate the oracles (*test failures*) or the user expectation (*field failures*). BCT builds I/O and interaction models. I/O models are Boolean expressions that constraint the values that can be assigned to parameters. Interaction models are finite state automata (FSA) that specify sequences of methods that can be invoked by a component when interacting with other components, like the FSA in Fig. 1. The rest of this section overviews BCT. Sections 3 and 4 describe model inference and failure analysis, respectively.

2.1 Building Behavioral Models

BCT builds models of component interactions by dynamically analyzing system executions and monitors interactions at the component interface level, without accessing the implementation details of the components. Monitoring the internal of components would cause a high overhead and produce huge traces making dynamic model generation infeasible for nontrivial systems. BCT can be implemented on top of several technologies that range from aspect oriented programming to software probes. We experimented with BCT both with Aspectwerkz [29] and with the probes available within the TPTP platform [30]. Both technologies can capture method invocations and references to the objects exchanged during computation.

Object references carry only limited information. To gather the additional information required to build useful models, we augment classic monitoring technologies with a graph-traversal algorithm that navigates the object graphs rooted in the references captured with monitoring. This algorithm accesses object attributes through reflection [31], visits the object graph with a breath-first visiting strategy,

1. Following the IEEE standard, we use the term *fault* to indicate an incorrect step, process, or data definition in a computer program, and the term *failure* to indicate the inability of a system or component to perform its required functions [28].

and includes mechanisms to detect and avoid cycles and multiple extraction of the same object attributes.

BCT builds models that represent component interaction sequences within a single execution flow. When monitoring concurrent systems, BCT distinguishes invocation sequences that are part of the same execution flow from interleavings that depend on the concurrent structure by separately recording sequences of method calls observed in different threads. The activity of each thread can be distinguished by recording thread identifiers, available in most modern platforms, like JVM and .NET, and operating systems, like Windows and Linux.

To analyze failures and diagnose faults, BCT needs models that approximate the correct behavior of software systems. BCT builds accurate models of correct behaviors by monitoring software systems when executed thoroughly and successfully. Natural scenarios to produce such models are system, regression, and acceptance testing, when systems are executed thoroughly, and oracles prune failed executions.

BCT infers models that summarize and generalize the observed behaviors incrementally. In this way, BCT avoids recording a large amount of execution traces and improves scalability. BCT generates two types of models that we refer to as *I/O* and *interaction* models. I/O models are Boolean expressions that are associated with the methods in the component interfaces and that generalize the relations among values exchanged during the software executions. For example, an I/O model `item.quantity` > 0 associated with a method `Cart.add(Item item)` specifies that the attribute `quantity` of parameter `item` passed to method `add` implemented by the class `Cart` held only positive values in the monitored executions. BCT generates I/O models with Daikon, an inference engine that can process data incrementally [32].

Interaction models are finite state automata that are associated with the methods in the component interfaces. They summarize the intercomponent invocations involved in the method execution. An interaction model associated with a method `m()` implemented by a component `C` indicates the methods of the other components of the system that `C` invokes when `m()` is executed. For example, the FSA shown in Fig. 1 indicates the sequences of method invocations that have been observed while monitoring the component interactions of method `order(Cart cart)` executed to create new orders: Method `order` extracts user information by calling `getUserByID()` and retrieves the cost of all items that are part of the order (method `getCost()`) before extracting the shipping address (method `getShippingInfo()`). BCT infers interaction models using kBehavior. We discuss in detail the incremental inference engine kBehavior in Section 3.

2.2 Analyzing Failures and Diagnosing Faults

BCT analyzes failures and diagnoses faults by comparing failed executions with I/O and interaction models built during successful executions. In particular, BCT identifies unexpected parameter values that violate I/O models and unexpected method calls that violate interaction models, filters false positives by exploiting suitable heuristics, and clusters the resulting events to diagnose the possible faults. We discuss the heuristics to filter false positives and the techniques to cluster related violations in detail in Sections 4 and 5.

BCT compares failed executions with I/O and interaction models by reexecuting the software after a failure occurrence. To reproduce failures that occur during testing, test designers can simply reexecute the test cases that led to failures. Reproducing failures experienced in the field may be difficult due to the lack of execution details. To overcome these problems, it is possible to augment applications with frameworks that capture the runtime data necessary to repeat executions [33].

BCT does not depend on the implementation environment, except for monitoring. In this paper, we report our experience with Java, but BCT can be extended to programs written with other languages by substituting the monitoring framework.

3 Synthesizing Models

In this section, we present the techniques that BCT uses to automatically generate I/O and interaction models from execution traces. As discussed in the introduction, we focus on the common case of component-based systems that do not offer complete access to source code of all components (gray-box components), and we analyze failures after systems have been released, for instance, during regression testing and field executions. Here, we summarize the key characteristics of our target problems, and we discuss the consequent constraints for the inference engines:

- We assume the availability of successful traces only, such as the traces produced during system and acceptance testing at release time. Thus, the inference engines can rely *only* on a set of *positive samples* to investigate failures observed after the system release. It cannot rely, for example, on negative samples, as assumed by some inference techniques [34].
- We assume that test designers do not have full control of the system under analysis due to the presence of gray-box components and cannot impose specific requirements on traces. Thus, the inference engines cannot rely on extra information on traces, such as hypothesis about the execution order of traces, as assumed by some techniques [35], or knowledgeable teachers that provide information on the admissibility of the traces recognized by the generated FSA, as assumed by other techniques [36].
- We assume that the monitored executions produce a large number of long traces. Thus, the inference engines must handle traces efficiently and cannot store the entire set of traces in the memory, as done, for example, by inference techniques that produce a complete Prefix Tree Acceptor before processing the traces [37], [38], [39].

We use *Daikon* to generate I/O models [40]. Daikon processes sets of samples composed of variables and associated values, and automatically generates predicates that are defined on the variables and are satisfied by all processed samples. For instance, given a set of samples with variable `item.quantity` associated only with positive values, Daikon can generate the predicate

TABLE 1
Information about Case Studies, Test Suites, and Traces Collected to Generate the Interaction Models

Case study identifier	Type	Num. test cases	Test suite description	Test suite developer	Traces		
					number	max length	avg length
X2Sql, Nano XML v4 to v5, CR_HD_1	RP	36	Integration suite implemented with JUnit	SIR researchers	2,466	5,780	60.36
X2Sql, Nano XML v4 to v5, NV_HD_1	RP						
X2Sql, Nano XML v4 to v5, SR_HD_1	RP						
Nano XML v4 to v5, CR_HD_1	RP	209	System test suite implemented with JUnit	Nano XML developers	303,126	539	1.57
Nano XML v4 to v5, NV_HD_1	RP						
Nano XML v4 to v5, SR_HD_1	RP						
Nano XML v4 to v5, XER_HD_1	RP						
Nano XML v4 to v5, all_f	RP						
Nano XML v5 to v5, all_f	FF				306,969	537	1.57
Tomcat 5.5.12 to 5.5.13	RP	121	Functional test suite for the Tomcat management functionality	the authors of this article	2,371	2,173	15.28
Tomcat 6.0.4	FF				28,782	4,350	6.11
Eclipse, WTP, EMF, GEF, JEM	RP	105	System test suite implemented with JUnit	WTP developers	15,870	3,345	5.37
Eclipse, EMF 2.2.1, WTP	FF						
Eclipse, GMF, EMF, OCL	RP	406	Unit and integration test suites for EMF and GMF implemented with JUnit	EMF and GMF developers	81,227	2,650	1.94
ProbeKit, Eclipse(chmod)	FF	10	Functional test suite for the Probekit instrumentation functionality	the authors of this article	2,789	781	10.3
ProbeKit, Eclipse(EMT64)	FF						

(Legend) Column Case study identifier *assigns identifiers to the case studies. Column* Type *specifies the type of the case study:* RP *for regression problems,* FF *for field failures. Column* Num test cases *indicates the number of test cases in the test suites that have been executed to collect traces. Column* Test suite description *provides a short description of the test suites: the terms* system, integration, unit, *and* functional test suites *indicate test suites that exercise the whole system, the integration between components, single components, and a specific functionality, respectively. Column* Test suite developer *indicates the sources of test suites. Column* Traces *provides information about the traces: the number of analyzed traces (column* number*), the longest analyzed trace (column* max length*), and the average length of the analyzed traces (column* avg length*). The max and average lengths of the trace are measured as the number of events in the traces.*

`item.quantity` > 0. Daikon filters predicates with statistical indexes that indicate the probability that predicates are coincidental. The statistical indexes depend on the kind of the predicate and are discussed in [40]. For instance, Daikon generates the predicate `item.quantity` > 0 only when the number of positive samples for `item.quantity` exceeds a parametric threshold that measures the probability of positive samples to be coincidental. Daikon works with successful traces only, does not require the traces to satisfy specific properties, and incrementally refines the set of inferred constraints while sequentially analyzing the traces, without loading the whole set of traces in memory. Thus, Daikon satisfies our constraints.

We observed that software components typically execute recurrent interactions that can be modeled with simple automata. We also observed that inference algorithms that recognize recurrent behaviors and process traces *incrementally* can compact many traces into small automata and keep in memory only the models, without storing traces once processed. Moreover, incremental inference engines are particularly well suited to process traces that are monitored after the generation of a first set of interaction models because they can process new traces *without reprocessing the entire set of traces*. Thus, we considered incremental algorithms to infer FSAs that model component interactions.

We process component interactions, and thus the inference heuristic should privilege recurrent behaviors over the semantics of states. Since the states of the automata do not necessarily correspond to relevant states of the components, the semantics of states is less relevant and often implicit, while the sequences of the transitions in the automata represent relevant behaviors of the components. Thus, we look for *identifying recurrent behaviors*, while many inference engines identify recurrent and likely equivalent states [37], [38], [41].

The incremental algorithms proposed in the literature focus on convergence rather than memory consumption. They infer a convergent series of FSAs, that is, FSAs that approximate increasingly better the model to be discovered, and do not bind the amount of traces that must be available at any time during the computation. The algorithm proposed by Oncina and Garcia requires all of the traces available in memory [34]. The algorithm proposed by Parekh et al. requires a large subset of traces available in memory and takes advantage of knowledgeable teachers [42]. Thus, these algorithms do not meet our needs.

We defined *kBehavior* to incrementally synthesize finite state automata that model component interactions. kBehavior does not guarantee convergence a priori, but bound memory needs by incrementally processing the traces and storing only the current automaton, while many popular algorithms store the entire set of traces in the form of a Prefix Tree Acceptor. The incremental processing of the traces takes advantage of the nature of the samples, which represent interactions among components, and thus usually repeatedly follow a small set of interaction patterns. The experiments reported in this paper suggest that kBehavior processes interaction samples efficiently (see Table 1 for data about the length of the traces analyzed by kBehavior) and does not suffer from lack of convergence. In the

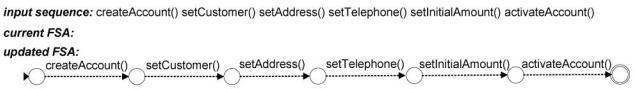


Fig. 2. The FSA that kBehavior generates by processing `seq1`. Dotted edges indicate the added transitions.

practical cases, the false positives that may derive from the lack of convergence are always filtered out by the heuristics presented in Section 4.

In the following, we first present kBehavior through an example, and then, we formalize the algorithm.

3.1 Example

In the rest of this section, we refer to the samples processed by kBehavior as *invocation sequences*, when we exemplify the algorithm with traces that contain method invocations, and as *strings*, when we present the theoretical elements of the algorithm. We do not use the term *traces* that describes data recorded at runtime, but is less appropriate to describe FSA inference algorithms.

Let us now consider the following example that consists of three invocation sequences obtained by recording the interactions between a client component and a bank account manager component:

seq1: createAccount() setCustomer() setAddress()
setTelephone() setInitialAmount() activateAccount()

seq2: createAccount() setCustomer() setAddress()
setMobilePhone() setInitialAmount()
activateAccount()

seq3: createAccount() setCustomer() setAddress()
setMobilePhone() addCustomer() addCustomer()
addCustomer() addCustomer() setInitialAmount()
activateAccount()

kBehavior processes the set of sequences incrementally, one after the other, without reaccessing sequences once processed. It starts with the first sequence (`seq1`) and builds an initial FSA that accepts the sequence. In some cases, the initial FSA maps the method calls to a linear sequence of transitions, as illustrated in Fig. 2. In other cases, kBehavior identifies simple recurrent patterns already in the initial sequence and models them in the FSA. We discuss this case in the next section.

When processing a new invocation sequence `seqi`, kBehavior extends the current FSA. First, it identifies the subautomata of the current FSA that accept subsequences of `seqi`. Then, it extends the current FSA by connecting the identified subautomata to obtain a new FSA that augments the language of the current FSA with `seqi`. We discuss how kBehavior identifies the subautomata and extend the current FSA in the next section.

For instance, while processing the second invocation sequence (`seq2`) starting from the current FSA shown in Fig. 3, kBehavior identifies two subautomata, *subautomaton1* and *subautomaton2*, that accept *subsequence1* and *subsequence2*, respectively. It also detects that in `seq2` the two subsequences are connected by the invocation of

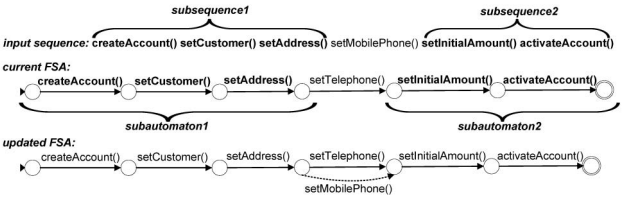


Fig. 3. The FSA that kBehavior generates by processing `seq2` after `seq1`. Dotted edges indicate the added transitions, bold cases indicate the matching subsequences.

method `setMobilePhone()`, while in the current FSA, the corresponding subautomata are connected by the transition `setTelephone()`. To include the new sequence in the language accepted by the FSA, kBehavior connects the identified subautomata with transitions that accept the invocations between the corresponding subsequences. In the example, kBehavior adds a transition `setMobilePhone()` between the final state of *subautomaton1* and the initial state of *subautomaton2*, as shown in Fig. 3.

kBehavior processes the subsequences of the current sequence recursively, to produce a compact FSA. As an example, Fig. 4 shows how kBehavior processes `seq3`. kBehavior identifies the subautomata *subautomaton1* that accepts the subsequence *subsequence1* and *subautomaton2* that accepts *subsequence2*, and connects the two subautomata with transitions that accept the four consecutive invocations of method `addCustomer()`. Before connecting *subautomaton1* to *subautomaton2*, kBehavior processes the subsequence with the four invocations to `addCustomer()` recursively and produces the looping automaton shown in Fig. 4. The shape of the loop (composed of two transitions) depends on the value of the parameter k of kBehavior and will be discussed in details in the next section.

3.2 kBehavior

Before defining kBehavior, we briefly recall some definitions.

A *Nondeterministic Finite State Automaton* is a five-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a nonempty finite set of states,
- Σ is a nonempty finite set of input symbols or input alphabet,
- $\delta : Q \times \Sigma \rightarrow \wp(Q)$ is the transition function that maps pairs of $\langle$state, symbol$\rangle$ into subsets of states,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting states.

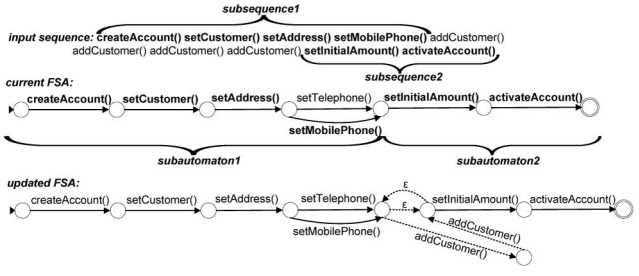


Fig. 4. The FSA that kBehavior generates by processing `seq3` after `seq2` and `seq1`. Dotted edges indicate the added transitions, bold cases indicate the matching subsequences.

If the function δ maps every pair $\langle \text{state, symbol} \rangle$ to a single state, that is, $\delta : Q \times \Sigma \to Q$, the automaton is *deterministic*. Σ^* denotes the set of all input strings over Σ. Given $\alpha \in \Sigma^*$, $|\alpha|$ denotes the length of α. Given $\alpha = \beta\gamma \in \Sigma^*$, we say that β is a *prefix* and γ is a *suffix* of α. δ^* denotes the straightforward extension of the transition function δ to strings:

$$\delta^*(q, \epsilon) = q, \text{ where } \epsilon \text{ is the empty string,}$$
$$\delta^*(q, a\alpha) = \delta^*(\delta(q, a), \alpha), \text{where } q \in Q, a \in \Sigma, \text{ and } \alpha \in \Sigma^*.$$

An input string α is *accepted* by a finite state automaton (either deterministic or not) if $\delta^*(q_0, \alpha) \cap F \neq \emptyset$. $L(A)$ denotes the *language* accepted by a FSA A, that is, the set of strings accepted by A.

Given a set of strings S, kBehavior builds a finite state automaton A, whose language $L(A)$ includes S. kBehavior starts with an empty FSA and processes the set of strings incrementally without revisiting the strings once added to the automaton. Given a string $s \in S$, the incremental step *extendFSA* modifies the input automaton A to extend $L(A)$ with the input string s. *extendFSA* initializes A and invokes the recursive step *addString* that looks for subautomata that accept substrings of s and connects the identified subautomata to include s in $L(A)$. *extendFSA* is invoked by both kBehavior to add a new string to the language accepted by current automaton and by *addString* when recursively processing a long substring.

In the following, we present the main steps of the algorithm kBehavior (Algorithm 1), the algorithm *extendFSA* (Algorithm 2), the algorithm *addString* (Algorithm 3), and the function *pattern* used in *addString* (Algorithm 4). We do not present the intuitive functions *merge* and *addTail* that are used in *addString*. The interested reader can refer to [43] for details. For the sake of simplicity, we present the algorithms referring to a deterministic mapping δ. The general algorithms can be defined by considering a nondeterministic function δ and by replacing single states with sets of states.

Algorithm 1. FSA kBehavior(S, k, k_ρ)
Require: a nonempty set of strings S that are processed by the algorithm
Require: two integers k and k_ρ, with $k > 0$ and either $k_\rho = 0$ or $k_\rho \geq k$
Ensure: an FSA A that accepts all strings in S: $S \subseteq L(A)$
1: $A \leftarrow \emptyset$
2: **for all** $s \in S$ **do**
3: $A \leftarrow$ extendFSA(A, s, k, k_ρ) *//extend A by incrementally adding the string s*
4: **end for**
5: **return** A

Algorithm 2. FSA extendFSA(A, s, k, k_ρ)
Require: an automaton A with initial state q_0
Require: a string $s = \langle s_1, \ldots, s_n \rangle$
Require: two integers k and k_ρ, with $k > 0$ and either $k_\rho = 0$ or $k_\rho \geq k$
Ensure: an automaton A' that accepts both the language accepted by A and s: $s \cup L(A) \subseteq L(A')$
1: **if** $A == \emptyset$ **and** s empty **then**
2: **return** an automaton with 2 states connected by an ϵ transition
3: **end if**
4: **if** $A == \emptyset$ **then**
5: $l \leftarrow min(k, n)$ *//select the first k symbols, if exist*
6: $A \leftarrow$ linear automaton with $l + 1$ states $(q_i | 0 \leq i \leq l)$ and l transitions labeled with the first l symbols of $s((\langle q_{i-1}, q_i \rangle, s_i) | 1 \leq i \leq l)$
7: $q \leftarrow q_l$ *//model generation continues from this state*
8: $s \leftarrow \langle s_{l+1}, \ldots, s_n \rangle$ *//the remaining portion of the string*
9: **else**
10: $q \leftarrow q_0$ *//model generation continues from the initial state, no changes to s*
11: **end if**
12: **if** s **not** empty **then**
13: $A \leftarrow$ addString(A, q, s, k, k_ρ) *//extend A from state q with string s*
14: **end if**
15: **return** A

Algorithm 3. FSA addString(A, q, s, k, k_ρ)
Require: an automaton $A = (Q, \Sigma, \delta, q_0, F)$
Require: a state $q \in Q$ that is the starting point to extend A
Require: a nonempty string $s = \langle s_1, \ldots, s_n \rangle$
Require: two integers k and k_ρ, with $k > 0$ and either $k_\rho = 0$ or $k_\rho \geq k$
Ensure: an FSA A' that extends A to accept s from state q
//Prefix step
1: $\beta \leftarrow$ the longest $\langle s_1, \ldots, s_{max} \rangle$ accepted by A from state q
2: **if** β **not** empty **then**
3: $s \leftarrow \langle s_{max+1}, \ldots, s_n \rangle$
4: **end if**
5: $ls \leftarrow n - max$
6: **if** s is empty **then**
7: **return** A
8: **end if**
9: $q \leftarrow \delta^*(q, \beta)$ *//accepting state reached when accepting β*
//Pattern step
10: $\langle \mu, \rho, \nu, q_\rho \rangle \leftarrow$ pattern(A, s, k, k_ρ)
//Merge step
11: **if** ρ is empty **then** *//no subautomaton of A accepts a substring of s*
12: $A_s \leftarrow$ extendFSA($\emptyset, s$)
13: $A \leftarrow$ addTail(A, q, A_s)
14: **return** A
15: **end if**
//ρ follows μ in s, and is accepted from state q_ρ of A. μ is not accepted by any subautomaton of A
16: $B \leftarrow$ extendFSA($\emptyset, \mu, k, k_\rho$) *//build an automaton that accepts μ*
17: $A \leftarrow$ merge(A, q, q_ρ, B) *//merge B with A at states q and q_ρ*
18: $q \leftarrow \delta^*(q, \mu\rho)$ *//accepting state reached when accepting $\mu\rho$*
19: **if** ν is **not** empty **then** *//ν is a tail of s still not accepted by A*
20: $A \leftarrow$ addString(A, q, ν, k, k_ρ)*//extend A according to the remaining trace ν*
21: **end if**
22: **return** A

Algorithm 4. $\langle \mu, \rho, \nu, q_\rho \rangle$ pattern(A, s, k, k_ρ)
Require: an automaton $A = (Q, \Sigma, \delta, q_0, F)$
Require: a string $s = \langle s_1, \ldots, s_n \rangle$
Require: two integers k and k_ρ, with $k > 0$ and either $k_\rho = 0$ (best matching mode) or $k_\rho \geq k$ (optimized mode)
Ensure: returns three strings μ, ρ and ν and a state q_ρ. If there exists a decomposition of $s = \mu\rho\nu$, such that ρ is a substring longer than or equal to k_ρ (optimized mode) or the longest substring longer than k (best matching mode) accepted by A from a state q_ρ, ρ is non empty; otherwise, ρ is empty and the value of μ, ν and q_ρ are not relevant.

1: $\rho \leftarrow$ empty
2: $length_\rho \leftarrow 0$ *//the length of* ρ
3: **for** $i = 1$ to n **do**
4: **for all** $q_j \in Q$ **do**
5: $l \leftarrow$ the longest $\langle s_i, \ldots s_{l+i} \rangle$ accepted by A from state q_j
6: **if** $l > length_\rho$ **and** $l \geq k$ **then**
7: $length_\rho \leftarrow l$
8: $\mu \leftarrow \langle s_1, \ldots s_{i-1} \rangle$
9: $\rho \leftarrow \langle s_i, \ldots s_{l+i} \rangle$
10: $\nu \leftarrow \langle s_{l+i+1}, \ldots s_n \rangle$
11: $q_\rho \leftarrow q_j$
12: **end if**
//if $k_\rho \neq 0$ *we use the* optimized *mode that returns a pattern when finding a string of length* $l \geq k_\rho$
13: **if** $l \geq k_\rho$ **and** $k_\rho \neq 0$ **then**
14: **return** $\langle \mu, \rho, \nu, q_\rho \rangle$ *//optimized mode*
15: **end if**
16: **end for**
17: **end for**
18: **return** $\langle \mu, \rho, \nu, q_\rho \rangle$

Finding subautomata that accept substrings of length one is straightforward and not very interesting. To avoid identifying trivial subautomata, kBehavior looks for subautomata that accept substrings with at least k symbols, where k is the main parameter of kBehavior. The parameter k_ρ is used to optimize the search of subautomata that accept substrings of the string s that is currently processed. When $k_\rho = 0$, kBehavior searches for a subautomaton that accepts the longest substring of s accepted by the current FSA, while when $k_\rho \geq k$, kBehavior terminates the search when it finds a subautomaton that accepts a substring of length greater or equal to k_ρ. In this way, kBehavior does not need to search through the whole automaton for every processed substring.

When invoked with an empty automaton, *extendFSA* initializes the automaton to the sequential automaton that accepts the first k symbols of s (line 6 in Algorithm 2). If s is longer than k, *extendFSA* removes the first k symbols from s and invokes the recursive step *addString* with the newly created linear automaton, the tail of the input string, and the last state of the automaton (line 13 in Algorithm 2). If s is shorter than k, *extendFSA* directly creates the sequential automaton that accepts s, removes all elements from s, and returns the newly created automaton. If s is empty, *extendFSA* initializes the automaton to the trivial automaton with two states connected by an ϵ transition and returns the newly created automaton (line 2 in Algorithm 2).

When invoked with a nonempty automaton, *extendFSA* simply invokes *addString* (line 13 in Algorithm 2) with the parameter q initialized to the initial state q_0 of A (line 10 in Algorithm 2). The parameter q is the state from which *addString* starts searching for a subautomaton that accepts a substring of the given string s.

addString processes an FSA A, a state q, and a string s in three main steps: Prefix (from line 1 to line 9 in Algorithm 3), Pattern (line 10 in Algorithm 3), and Merge (from line 11 to line 18 in Algorithm 3).

3.2.1 Step Prefix of addString

In the *Prefix* step, *addString* identifies the longest prefix β of s accepted by A starting from state q (line 1), removes β from s (line 3), and advances the state q to the final state of the subautomaton that accepts β (line 9). In this way, *addString* recognizes the longest prefix of s already included in A starting from q, and tries to extend A beyond the prefix to accept the tail of s. For example, when *addString* processes the second trace in Fig. 3, it identifies the prefix `createAccount(), setCustomer(), setAddress()` accepted by the current FSA and proceeds with the tail `setMobilePhone(), setInitialAmount(), activateAccount()`. If the tail of s is empty, s is already accepted by A, and *addString* returns A (line 7 in Algorithm 3). Otherwise, *addString* tries to identify subautomata that accept substrings of s (step *Pattern*).

3.2.2 Step Pattern of addString

In the *Pattern* step, *addString* looks for a substring ρ of s both longer than k and accepted by a subautomaton of A. The Pattern step consists of a call to function *pattern* (line 10 in Algorithm 3) that is detailed in Algorithm 4. (The algorithm is not optimized; faster search strategies can be considered to improve the performance of the algorithm.)

The pattern function can work in two modes. In *best matching* mode ($k_\rho = 0$), that corresponds to lines 3-12 in Algorithm 4, the pattern function scans s sequentially and looks for the longest substring accepted by a subautomaton of A. This mode can be very expensive: In the worst case, when the longest substring is the tail of s, the pattern function scans the whole string s and searches for all possible subautomata that accept substrings of s. In *optimized* mode, the pattern function uses a value for the parameter $k_\rho \geq k$ and stops looking for substrings accepted by a subautomaton of s when it finds a substring not shorter than k_ρ. In a nutshell, the algorithm stops as soon as it finds the first good-enough match. The halt criterion that characterizes the optimized search strategy corresponds to lines 13-15 in Algorithm 4.

Fig. 5 shows that the optimized mode not only improves performance, but may produce better results than in the best matching mode. Fig. 5a shows *addString* working in best matching mode with $k = 2$. *addString* correctly identifies the longest substring accepted by a subautomaton of the current FSA as the tail `m3 m3 m2` of the input string and connects the subautomaton that accepts the head of the string to the subautomaton that accepts the tail of the string, as informally discussed in the former section. Thus, in the best matching mode, kBehavior misses many similarities between the two paths connecting the two subautomata.

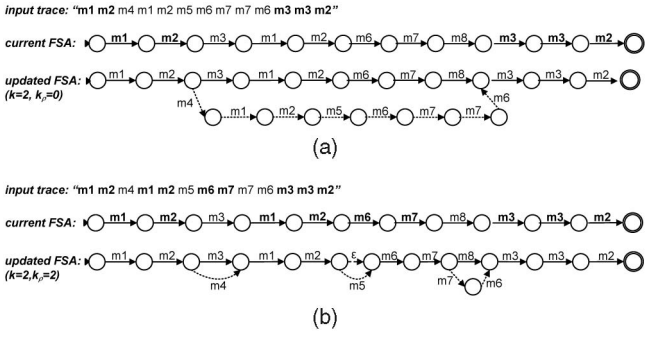


Fig. 5. The effects of (a) best matching and (b) optimized behavioral pattern identification in kBehavior. Dotted edges indicate the added transitions, bold cases indicate the matching substrings.

Fig. 5b shows *addString* working in optimized mode with $k_\rho = k = 2$. *addString* incrementally identifies three subautomata that accept substrings of the input string and produces a more compact FSA.

BCT uses the optimized mode of kBehavior since in this mode kBehavior performs better and produces more compact automata than in best matching mode. Our experience in monitoring the interactions between software components with kBehavior indicates $k_\rho = k = 2$ and $k = 2$, $k_\rho = 4$ as good choices for these two parameters. The rational for such small and efficient values for k and k_ρ is grounded in the small average size of the automata that model component interactions. In fact, when monitoring the execution of component interfaces, we usually observe small sets of interactions that are generalized by small automata. When used in other domains characterized by large automata, good choices of the parameter values for kBehavior may be different.

3.2.3 Step Merge of addString

The step Pattern of *addString* decomposes s into three substrings $\mu\rho\nu$ and finds the state q_ρ, which is the root of a subautomaton of A that accepts ρ. If ρ is empty, there is no subautomaton of A that accepts a substring of s with at least k symbols. Thus, the *Merge* step of *addString* simply connects q to an automaton that accepts s (lines 11-15 in Algorithm 3). If ρ is not empty, the *Merge* step of *addString* connects q to q_ρ by means of an automaton B that accepts μ, and then continues processing ν (lines 16-18 in Algorithm 3).

The Merge step of *addString* generates B by recursively calling *extendFSA* with μ and an empty automaton as parameters (line 16 in Algorithm 3). It includes B in A by merging q with the initial state of B, and the final state of B with q_ρ (call of the *merge* function at line 17 in Algorithm 3). For example, when *addString* processes the second trace as shown in Fig. 3, *addString* adds a transition `setMobilePhone()` between the subautomaton that accepts the prefix `createAccount() setCustomer() setAddress()` and the one that accepts `setInitial Amount() activateAccount()`.

Once the current FSA has been extended with B, *extendFSA* invokes *addString* with the new FSA, the state reached after accepting $\mu\rho$ from q, and ν, to complete processing s (line 20 in Algorithm 3).

3.3 Termination and Complexity

Given a set of samples S, kBehavior produces an automaton that accepts all samples in S by construction. The recursive algorithm kBehavior always *terminates*, in fact, it removes at least k symbols from s at each recursive step. This happens when kBehavior does not find any substrings of the input string s already accepted by the current automaton, and thus removes only k symbols from s (line 8 in Algorithm 2) after adding a new branch of length k to the automaton (line 6 in Algorithm 2). Thus, kBehavior analyzes a trace of length l in at most $\frac{l}{k}$ recursive steps. In general, kBehavior advances of $p \geq k$ symbols at each step, and thus processes a string of length l in $\frac{l}{p}$ steps.

3.3.1 Complexity Analysis

The algorithm includes several recursive calls that challenge the computation of the complexity. To estimate the average and worst-case complexity, we introduce several over-approximations. Thus, we compute upper bounds to the average and worst-case complexity, but we cannot estimate the degree of approximation. Here, we report and discuss the final results. The detailed computation of the complexity is given in [43].

When extending an automaton with a new trace, the complexity of kBehavior depends on the number q of states in the automaton to be extended with a new trace and the length l of the trace to be processed.

The best-case complexity corresponds to the case of a trace already accepted by the current automaton. In this case, the trace is processed in linear time ($O(l)$ operations).

The worst-case complexity depends on the kind of inferred automaton. When inferring a deterministic automaton, the overapproximated worst-case complexity is $O(ql^2 + l^3)$. This result suggests a high impact on the length of the string. However, our empirical experience indicates that the algorithm can also process long strings efficiently. This depends on the characteristics of the strings that we analyze: Traces obtained by monitoring software components show a good regularity that kBehavior exploits to reduce the number of extensions needed to accept the new traces.

When inferring a nondeterministic automaton, the over-approximated worst-case complexity is $O(q^2l^2 + l^4 + ql^3)$. Fortunately, component interactions are mostly deterministic, and nondeterminism occurs only rarely when deterministic decision internal to a component is masked by the interfaces or in some domains, such as distributed systems. To confirm this intuition, we measured the degree of nondeterminism (ND) of the automata that we derived in the empirical validation presented in Section 6. We define the degree of nondeterminism ND as the ratio between the number of nondeterministic events that can be accepted by the automata from any state (NDT) and the number of events that can be accepted by the automata from any state (AE)

$$ND = \frac{|NDT|}{|AE|}.$$

In the formula above, $\|$ is the cardinality of a set and NDT and AE are defined as follows: Given the set IA of the automata that we inferred in our empirical validation, NDT

is the number of nondeterministic events that can be accepted by the automata from any state

$$NDT = \{e | e \neq \epsilon, \exists A = (Q, \Sigma, \delta, q_0, F) \in IA, \exists\, q \in Q\ s.t.\ |\delta(q, e)| > 1\},$$

and AE is the set of events that can be accepted by the automata from any state

$$AE = \{e | e \neq \epsilon, \exists\ A = (Q, \Sigma, \delta, q_0, F) \in IA, \exists\, q \in Q\ s.t.\ |\delta(q, e)| > 0\}.$$

Intuitively ND measures the likelihood that an event is accepted by a nondeterministic transition. In our empirical studies, ND is equal to 12.66 percent, which is a small fraction of all of the events.

To estimate the average complexity, we observe that, at every step, kBehavior splits a string into four subsequences, $prefix$, μ, ρ, and ν. We estimate the average complexity by considering a splitting that produces subsequences of the same length $\frac{l}{4}$. The resulting overapproximated average complexity is $O(ql + l^2)$. The strong dependency on the length of the string is inherited from the worst-case complexity and is not confirmed by our experiments that indicate better values, as discussed above. The even division between the four components of s is not confirmed by our experiments either, where $prefix$ and ρ are usually much longer than μ. Different distributions for the lengths of the subsequences could improve the complexity results, but we do not have enough elements to consider different distributions of lengths.

4 Analyzing Failures to Diagnose Faults

BCT analyzes failures and diagnoses faults by comparing failed executions with the I/O and the interaction models built during successful executions. BCT first identifies the events and values that belong to failed executions and violate the models, and then analyzes such events and values to identify likely failure causes. Model violations can correspond either to faulty elements or to successful executions not experienced in the past (false positives). Violations that correspond to faulty elements may belong to large sets of violations, all related to the same fault.

Manually inspecting all violations to distinguish false from true positives (TP) can be extremely time-consuming and can reduce the applicability of BCT [7], [8]. BCT overcomes this problem by introducing some simple but effective heuristics that eliminate most model violations that correspond to false positives.

Large sets of violations that correspond to few faults derive from cascades of differences between failed executions and models. Such cascades of differences are triggered by few relevant model violations. Inspecting a flat set of violations is tedious and error prone since single violations often provide only partial information about the failure. BCT addresses this problem by automatically aggregating related violations to provide a comprehensive view of the failure causes. The aggregated views are easier to inspect and analyze.

In the following, we informally present the techniques that we defined to filter false positives and aggregate related anomalies. Section 5 formally presents our clustering strategy to aggregate related anomalies. We report experimental data that illustrate the effectiveness of filters and aggregators in Section 6.

4.1 Filtering False Positives

We automatically distinguish violations that are likely related to faults from false positives with a simple but effective heuristic: *Model violations that occur both during successful and failed executions are likely related to new software behaviors, while model violations that occur only during failed executions are most likely anomalies*. Thus, we automatically discard model violations that occur during both successful and failed executions, and we analyze violations that occur only during failed executions. This heuristic was effective in filtering false positives in our case studies. The heuristic is inspired by the fault localization techniques proposed by Liu and Han [21] and Liblit et al. [22].

This heuristic requires the availability of both failing and successful executions. These are available, for example, when BCT investigates regression faults. Executing a regression test suite to validate a new version of a software system results in both failing and successful executions, and both kinds of executions can contain model violations that may derive either from faults (in failing executions only) or from changes implemented by developers (usually both in failing and successful ones). BCT analyzes failing executions, and uses successful executions to identify false positive with the heuristics introduced in this section.

4.2 Discovering Relations between Violations

Usually violations of models do not occur in isolation, but a single problem can be related to several model violations. For instance, an erroneous value can violate an I/O model and then generate an exception that violates many interaction and I/O models until the program either recovers from the exception or fails completely. Often, single violations provide only partial information about the fault, while sets of related violations can better indicate the nature and the localization of the fault. It is thus important to cluster sets of related violations.

We empirically observed that *violations related to the same fault are usually detected in methods executed from a (close) common ancestor method in the hierarchy of dynamic method invocations*. This happens because when a method executes an operation that fails, the failure is usually followed by a sequence of anomalous actions that violate other models while executing the remaining operations in the method. Thus, we cluster violations according to their distance from a common ancestor method.

To identify clusters of violations, we build an *anomaly graph* that represents the likely cause-effect relation between anomalous events, which are events that violate I/O or interaction models. We say that an anomalous event a causes an anomalous event b if b is a direct or indirect consequence of a. For instance, a `null` value that violates an I/O model can cause a `NullPointerException` when the application invokes a method on the `null` object, which, in turn, can generate an anomalous missing method invocation that violates an interaction model.

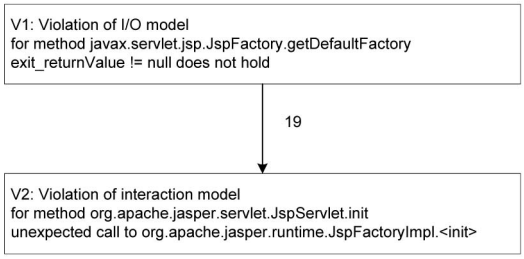


Fig. 6. The anomaly graph corresponding to the problem experienced on Tomcat 6.0.4. The weight on the edge represents the distance of the two violations in the dynamic call tree.

Nodes of the anomaly graph represent anomalous events, and edges represent likely cause-effect relations between events. Nodes are labeled with model violations, while edges are weighted according to the distance of the violations in the dynamic call tree, as discussed in detail later.

Fig. 6 shows a simple example of an anomaly graph. The anomaly graph includes a single connected component that represents the cause-effect relation between the two experienced model violations. Anomaly graphs are composed of one or more connected components. A connected component is an acyclic graph that models a set of anomalous events that are likely related to the same runtime problem. Multiple connected components represent multiple runtime problems, all detected in the same failed execution and each one related to one or more anomalies. An acyclic connected component always has at least one root node. The root node of a connected component in an anomaly graph represents the model violation that has likely caused the violations represented by the nodes in the same component. A component with more than one root indicates that BCT identifies several potential causes of the runtime problem.

Failures that depend on multiple faults, for instance, many failures that derive either from state-based faults or from the rare combination of multiple events, generate anomaly graphs that can be heuristically refined into multiple connected components, each corresponding to the cause of a failure. The heuristic refinement is informally introduced later in this section and formally described in Section 5.

BCT builds an initial anomaly graph from the dynamic call tree of a failed execution and from the set of model violations revealed during the failed execution. Fig. 7 shows an example of a dynamic call tree that corresponds to a failure in Tomcat. BCT then refines the initial anomaly graph by removing likely useless edges so that the remaining edges highlight the different faults that affected the program execution. A refined anomaly graph improves the initial anomaly graph by 1) partitioning the anomalies into multiple components that can isolate false positives from useful anomalies, and 2) partitioning unrelated anomalies into clusters of related anomalies, thus producing a clearer representation of the failure. Partitioning the anomalies into multiple components that can isolate false positives is relevant when analyzing medium/large size anomaly graphs, where several false positives can reduce the interpretability of the graph. Partitioning unrelated anomalies into clusters of related ones improves the conceptual

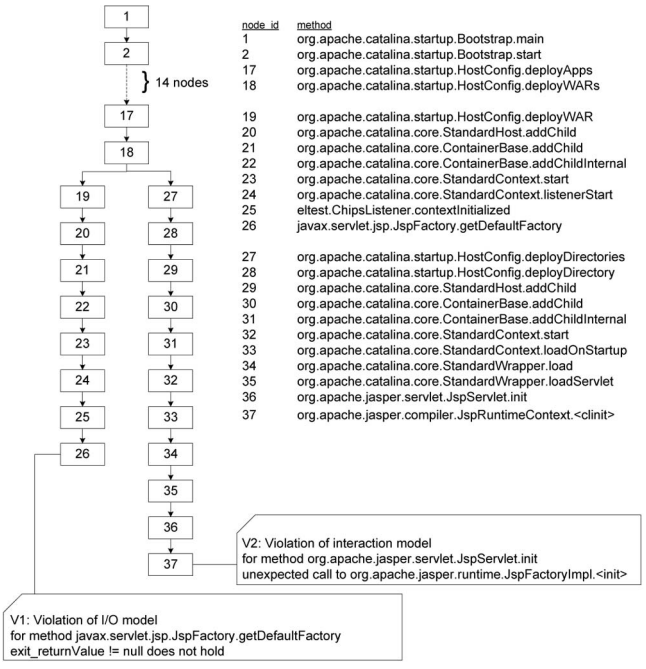


Fig. 7. The dynamic call tree corresponding to a failure in the Tomcat 6.0.4 application server (nodes from 3 to 16 are omitted for brevity).

view of the possible causes of the failures, thus significantly reducing the time required to interpret them.

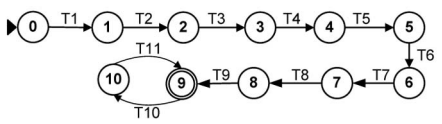
We informally present how BCT builds the anomaly graph by first discussing how BCT extracts the dynamic call tree from a failed execution, then how it generates the initial anomaly graph, and finally how it refines the initial anomaly graph into the final anomaly graph. We formalize the refinement strategy in Section 5.

4.2.1 Dynamic Call Tree

Here, we quickly recall the definition of dynamic call tree (DCT) [44]. A dynamic call tree is a tree that represents a program execution, in our case, a failed execution. Nodes represent methods, directed edges represent method invocations. A node A is connected to a node B if method A has invoked method B in the considered execution. We define the distance between two nodes n_i and n_j as the minimum number of edges to be traversed to move from n_i to n_j, ignoring the direction of the edges. Hereafter we refer to the distance between two nodes of the dynamic call tree as *DCT distance*. For example, Fig. 7 shows the dynamic call tree corresponding to a known failure of Tomcat 6.0.4. The DCT distance between nodes 26 and 37 is 19 since we need to traverse at least 19 edges to move from node 26 to node 37.

4.2.2 Initial Anomaly Graph

To analyze a failed execution and identify the faults, BCT builds an initial anomaly graph. Anomaly graphs indicate the relation between model violations that have been revealed during the failed execution and that have not been identified as false positives by the heuristic discussed before. The nodes of the initial graph represent model violations. The edges indicate the order of occurrence of violations. An edge $\langle A, B\rangle$ connects a violation A that occurred before B in the execution. We sort violations according to their timestamps.



transition	corresponding method
T1	javax.servlet.GenericServlet.init((Ljavax.servlet.ServletConfig;)V)
T2	java.io.File.exists(()Z)
T3	java.io.File.canRead(()Z)
T4	java.io.File.canWrite(()Z)
T5	java.io.File.isDirectory(()Z)
T6	javax.servlet.jsp.JspFactory.<clinit>(()V)
T7	javax.servlet.jsp.JspFactory.<init>(()V)
T8	javax.servlet.jsp.JspFactory.setDefaultFactory((Ljavax.servlet.jsp.JspFactory;)V)
T9	java.net.URLClassLoader.getURLs(()[Ljava.net.URL;)
T10	java.net.URL.getProtocol(()Ljava.lang.String;)
T11	java.net.URL.getFile(()[Ljava.lang.String;)

Fig. 8. The interaction model violated in $V2$: From state 5, Tomcat invokes the method corresponding to transition $T7$ before invoking the method corresponding to transition $T6$.

Edges are annotated with the DCT distance between the nodes that correspond to the violations.

Fig. 6 shows the anomaly graph derived from the Tomcat dynamic call tree of Fig. 7 that contains two model violations at nodes 26 and 37. In this case, the initial graph is a single acyclic connected graph that provides enough information to identify the fault. The graph corresponds to a known start-up problem of Tomcat 6.0.4. The problem occurs when Tomcat is running with web applications that execute JSP factories during initialization. The fault is in the Tomcat application server and is a missing variable initialization that causes the erroneous propagation of a `null` value within the system.[2] Although easy to describe and fix, this fault is hard to diagnose. In fact, it was discovered in release 6.0.4 and diagnosed and removed only after six releases, in release 6.0.10. The initial anomaly graph of Fig. 6 immediately indicates the problem and its cause. Node $V1$ corresponds to the violation of the I/O model $\mathrm{exit\_returnValue} != \mathrm{null}$ that is associated with method `JspFactory.getDefaultFactory()`. This violation indicates a failure in obtaining a factory object which accurately represents and localizes the initialization problem in Tomcat. Node $V2$ corresponds to the violation of the interaction model shown in Fig. 8: In state 5, method `JspServlet.<init>(...)` invokes method `JspFactory.<init>(...)` without invoking the static class constructor `JspFactory.<clinit>(...)` that would correctly initialize the variable. Method `JspServlet.<init>(...)` is identified by the transition $T7$, while method `JspFactory.<clinit>(...)` is identified by the transition $T6$. This model violation clearly indicates the nature of the failure caused by the initial problem: Tomcat does not invoke the static constructor of class `JspFactory` and thus fails.

The initial anomaly graph is immediately useful to diagnose the fault when it represents a small number of related violations, as in the Tomcat example discussed above. Large graphs, like the one in Fig. 10 that corresponds to a failure of Eclipse 3.3, do not immediately indicate the faults and need to be refined (this problem has been investigated as part of our case studies and is indicated under the entry `Eclipse,GMF,EMF,OCL` in Table 1). If the information provided by the initial anomaly graph is not clear, developers can inspect the anomaly graph automatically refined as follows.

2. http://issues.apache.org/bugzilla/show_bug.cgi?id=40820.

4.2.3 Refined Anomaly Graph

BCT refines the initial anomaly graph by removing the edges that connect distant violations, and thus should not correspond to semantically relevant relations, according to our heuristic.

BCT identifies a weight *BestWeight* that indicates violations that are too distant and thus likely unrelated, and removes all edges with weight greater than or equal to *BestWeight*. The distance *BestWeight* that distinguishes semantically relevant relations from irrelevant relations strongly depends on the graph topology. Inappropriate values for *BestWeight* reduce the effectiveness of the approach. Values too small can fragment the graph, wasting important information; values that are too large do not significantly reduce the graph that remains hard to interpret.

We defined an incremental clustering strategy to identify suitable values for *BestWeight* [45]. Intuitively, our incremental clustering strategy works as follows: BCT considers the sequence of anomaly graphs obtained from the initial graph by incrementally removing edges of decreasing weight, starting from the edges with the highest weight, until the edges with the lowest weight. BCT then measures the average *coupling* of the connected components in each anomaly graph. Intuitively, the coupling indicates the degree of focus of the connected components on single problems. Isolated nodes are associated to the best coupling ($coupling = 0$).

BCT considers in the sequence of graphs only those with both a different number of weakly connected components[3] and a better coupling (the concept of coupling is formally defined in the next section). For instance, if removing the edges with weight greater or equal to M generates a graph g and removing the edges with weight greater or equal to $M-1$ generates a graph g' with the same number of connected components as g, the graph g' does not belong to the sequence. BCT ignores graphs with the same number of connected components because we are interested in clusters of violations and not just in removing edges. Similarly, BCT does not consider anomaly graphs where the coupling does not improve because we are interested in clusters that better focus on the single problems that have been detected, and not just in any cluster.

When considering the sequence of graphs generated by BCT, we notice that initially the coupling decreases fast, but at a given point the decreasing rate becomes slow. We experimentally noticed that the different decreasing rate of the coupling is related to the relevance of the cause-effect relations between nodes. When the coupling decreases fast, BCT removes "heavy" edges, i.e., edges labeled with a high distance value, that are likely to connect unrelated anomalies. When the coupling decreases slowly, by removing further edges BCT is likely to disconnect related components, thus missing useful information. BCT computes the value of $BestWeight$ referring to the maximum

3. A weakly connected component is a maximum subgraph where all pairs of nodes are weakly connected, that is, they are connected in the associated undirected graph [46]. Hereafter, we will refer to weakly connected components simply as connected components.

variation between the coupling of consecutive graphs, which intuitively corresponds to select the anomaly graph where the heavy edges have been removed but the useful edges are still in the graph. This rational is shared with popular clustering algorithms based on *within clustering dispersion* [45].

5 Formalizing the Aggregation of Related Anomalies

In this section, we present the clustering strategy: We first define the concepts of anomaly graphs, weakly connected components, and coupling; we then show how to generate a refined sequence of anomaly graphs from the initial anomaly graph; we discuss a technique to identify the value of $BestWeight$ that identifies the anomaly graph returned by the clustering strategy and we present the visual interpretation of the clustering strategy.

5.1 Anomaly Graphs, Weakly Connected Components, and Coupling

An *anomaly graph* is a weighted directed graph $G = (N, E, m_W)$, where

- $N = \{n_1, \dots n_K\}$ is a finite nonempty set of nodes,
- $E \subseteq N \times N$ is a set of directed edges,
- $m_W : E \to \mathbb{N}_0$ is a mapping that associates weights to edges.

When the weight function is clear from the context, we indicate the value of $m_W(e)$ with e_w.

An anomaly graph $G = (N, E, m_W)$ is *weakly connected* if for any pair of nodes $n_a, n_b \in N$, there exists an undirected path from n_a to n_b that is a sequence $\langle \{n_a, n_1\}, \{n_1, n_2\}, \dots, \{n_{L-1}, n_L\}, \{n_L, n_b\} \rangle$ such that:

- $(n_a, n_1) \in E \vee (n_1, n_a) \in E$,
- $(n_i, n_{i+1}) \in E \vee (n_{i+1}, n_i) \in E, \forall\, i = 1 \dots L-1$,
- $(n_L, n_b) \in E \vee (n_b, n_L) \in E$,

The initial anomaly graph obtained by BCT from the dynamic call tree of the failing execution as described in the previous section is weakly connected by construction.

During the refinement process, we remove edges and disconnect the graph. In a disconnected graph, we can identify a finite set of weakly connected components that we formally introduce to define the concept of coupling of anomaly graphs.

A *weakly connected component* CC of G is a subgraph of G such that:

- CC is a weakly connected graph.
- It is not possible to add a node in G to CC and preserve weak connectivity for CC.

We can now define the concept of coupling of anomaly graphs starting from the definition of coupling of weakly connected components.

The *coupling of a weakly connected component* $CC = (N, E, m_W)$ is the average weight of its edges:

$$coupling(CC) = \begin{cases} \frac{\sum_{e_i \in E} e_{i_w}}{|E|} & for\ |E| > 0 \\ 0 & for\ |E| = 0. \end{cases}$$

The *coupling of an anomaly graph* G composed of the set of weakly connected components $\{CC_1, \dots, CC_m\}$ is the average coupling of the connected components:

$$coupling(G) = \frac{\sum_{i=1 \dots m} coupling(CC_i)}{m}.$$

5.2 Generating the Refined Sequence of Anomaly Graphs

We can now define the refinement sequence of anomaly graphs as the sequence of graphs obtained from the initial graph by incrementally removing the edges of highest weight. We use the refinement sequence of anomaly graphs to compute $BestWeight$, which is essential for our clustering strategy.

Let $\mathbb{G}raph$ be the set of weighted directed graphs. Given an anomaly graph $G = (N, E, m_W) \in \mathbb{G}raph$ and a function $maxW : \mathbb{G}raph \to \mathbb{N}_0$ that returns the maximum of the weights associated with the edges of a graph $G \in \mathbb{G}raph$ (0 for graphs with no edges), we define a function $next : \mathbb{G}raph \to \mathbb{G}raph$ that refines anomaly graphs as follow:

$next(G) = G'$, where $G' = (N, E', m'_W)$, with $E' = \{e \in E | m_W(e) < maxW(G)\}$ and m'_W is the restriction of m_W to E'.

Given an anomaly graph G, we can obtain a *sequence of graphs* SG as follows: $G_0 = G, G_1 = next(G_0), G_2 = next(G_1), \dots, G_k = next(G_{k-1})$. The sequence SG is finite and terminates with a graph with no edges and $maxW(G_k) = 0$.

We now refine the sequence by removing the graphs that either do not have a better coupling or do not include more weakly connected components than their predecessors since these graphs do not provide additional information to identify runtime problems with respect to the graphs in the refined sequence.

Let $countCC : G \to \mathbb{N}_0$ be the function that counts the number of weakly connected components in a graph. The sequence of graphs $SG = G_0, G_1, \dots, G_k$ can be refined into the longest sequence $RG = RG_1, \dots, RG_p$ that satisfies the following properties:

- $\forall i = 1, \dots p \exists$ one and only one j, such that $RG_i = G_j$ (we write $corr(RG_i) = j$) (RG is a subset of SG),
- if $i < j, corr(RG_i) < corr(RG_j)$ (RG preserves the order of SG),
- $\forall i = 1, \dots p-1, \quad countCC(RG_i) < countCC(RG_{i+1})$ (graphs in RG contains a strictly increasing number of connected components) and $coupling(RG_i) > coupling(RG_{i+1})$ (graphs in RG have a strictly decreasing coupling).

5.3 Identifying *BestWeight* and the Corresponding Anomaly Graph

To define $BestWeight$ we introduce the functions $gain$ and $trend$ that measure the change of coupling within a refined sequence RG of graphs. The coupling of the graphs that belong to a refined sequence RG is strictly decreasing and is 0 for the last element of the sequence (a graph with no edges.) The graphs that belong to the head of the refined sequence aggregate sets of nodes that correspond to different problems into single connected components. The graphs that belong to the tail of the refined sequence break

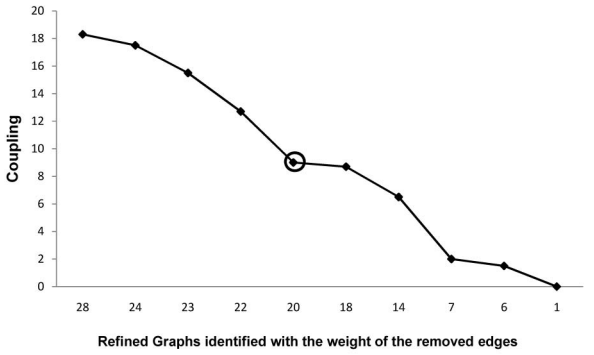


Fig. 9. The trend for the Eclipse 3.3 case study.

sets of nodes that correspond to the same problems into distinct components. We noticed experimentally that the graph that provides the best information about the problems under investigation corresponds to a sudden variation in the change of coupling of the graphs in the sequence. The functions $gain$ and $trend$ compute the changes of coupling and allow us to identify the sharpest variation that corresponds to $BestWeight$ and identifies the anomaly graph of best usage for developers. Function $gain$ is the ratio between the differences of coupling and the differences of weights of consecutive graphs in a refined sequence of graphs RG.

Let $RG = RG_1, \ldots, RG_n$ be a refined sequence of graphs. For all pairs of consecutive graphs RG_i with $MaxW(RG_i) = w_i$ and RG_{i+1} with $MaxW(RG_{i+1}) = w_{i+1}$, the $gain$ from RG_i to RG_{i+1} is

$$gain(RG_i, RG_{i+1}) = \frac{coupling(RG_i) - coupling(RG_{i+1})}{w_i - w_{i+1}}.$$

We denote $gain(RG_i, RG_{i+1})$ as $gain_{i,i+1}$. Intuitively, function $gain$ represents the increment of the coupling between consecutive graphs in the sequence, and corresponds to the slope of the line connecting the points with coordinate $(w_i, coupling(RG_i))$ and $(w_{i+1}, coupling(RG_{i+1}))$ in the Cartesian plan.

Given a sequence of gains obtained from a refined sequence of graphs, the $trend$ in gain at step i is the difference of gains between the considered and the previous step in the sequence.

Let $gain_{1,2}, gain_{2,3}, gain_{3,4}, \ldots gain_{n-1,n}$ be the sequence of gains computed for a refined sequence of graphs RG_1, $RG_2, \ldots, RG_n$,

$$trend_i = |gain_{i,i+1} - gain_{i-1,i}|, \forall i \in 2, \ldots n-1.$$

The difference between two consecutive values of $gain$ can be positive or negative; the absolute value represent the degree of change independently from the direction.

$BestWeight$ is the weight that corresponds to the highest value of trend.

Let $trend_2, trend_3, \ldots trend_{n-1}$ be the sequence of trends computed for a refined sequence of graphs $RG_1, RG_2, \ldots$, RG_n,

$$BestTrend = k | trend_k \geq trend_i, \quad \forall i = 2 \ldots n-1,$$
$$BestWeight = w_{BestTrend}.$$

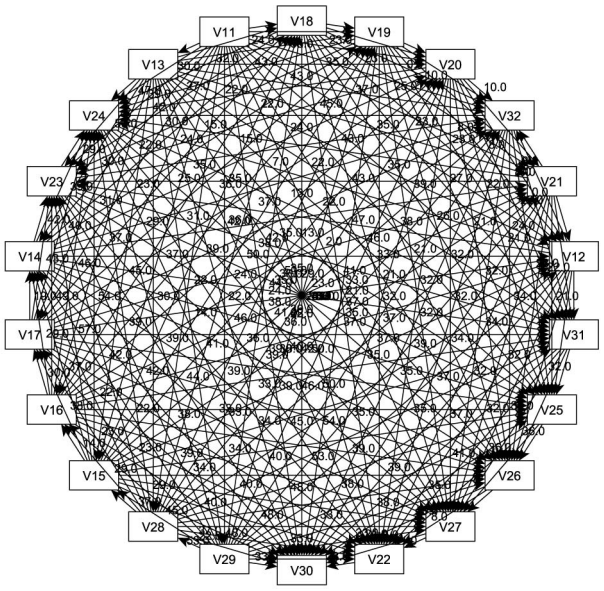


Fig. 10. The initial anomaly graph corresponding to the problem experienced with Eclipse 3.3.

5.4 Interpreting the Clustering Strategy Visually

We visually illustrate our strategy by plotting the values of the coupling with respect to the refined sequence of anomaly graphs. The graphs in the sequence are identified with the value of the weight of the removed edges. Fig. 9 shows the result for the Eclipse 3.3 case study that we discuss in more details below. The gain corresponds to the slope of the lines that connect the points. The best trend $BestTrend$ is the point of highest difference between consecutive gains and corresponds to the sudden change from a big to a small gain indicated by the circled point in Fig. 9 (intuitively, the gain and the trend would correspond to the first and second derivatives of the coupling function in the domain of continuous functions) $BestTrend$ identifies the weight $BestWeight$. The configuration associated with $BestWeight$ corresponds to a partitioning of the detected violations that well represents the problems occurred in the failing execution. It consists of 11 weakly connected components that represent the 11 sets of related violations. Fig. 10 shows the initial (connected) anomaly graph, while Fig. 11 shows the anomaly graph that is identified by $BestWeight$ and is composed of 11 disconnected components.

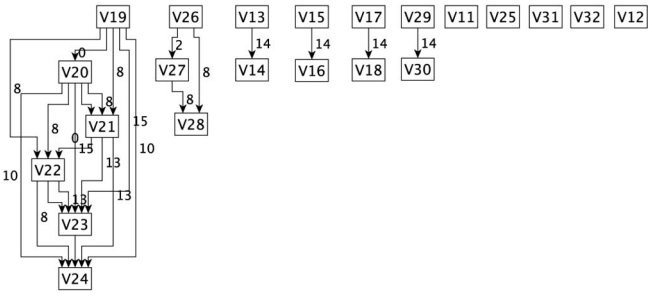


Fig. 11. The anomaly graph that corresponds to the problem experienced with Eclipse 3.3. This graph is derived from the initial anomaly graph shown in Fig. 10.

As discussed in the previous sections, faults usually correspond to cascades of anomalies, that is, sets of anomalies closely related in the dynamic call tree, while false positives often correspond to isolated anomalies. Thus, we examine the weakly connected components in the final anomaly graph sorted by size, from the biggest to the smallest. In the Eclipse 3.3 case study, the relevant connected components that describe the failing execution are the two largest ones, which represent two runtime problems detected by BCT. The remaining nine weakly connected components correspond to false positives and can be ignored by the developers, who can understand the problem by looking at the first two components only.

The problem of the Eclipse 3.3 case study is related to a fault that prevents the execution of an undo operation. The first component of the refined anomaly graph indicates an anomalous `null` value returned by method `NotificationImpl.getNewValue()` at node $V20$, while the second component indicates an anomalous `null` value returned by method `NotificationImpl.getOldValue()` at node $V26$. These are the two methods that are executed by the application to complete an undo. The clustering strategy correctly divides the initial anomaly graph into components and correctly ranks them to understand the problem: The first component indicates the call that returns an unexpected `null` value, rather than the old value required by the undo operation; the second component indicates the call that returns an unexpectedly `null` value, rather than the new value required by the undo operation. The two clusters with highest rank carry enough information to understand the cause of the failure and diagnose the faults, while all lower ranked components represent spurious violations that carry information irrelevant for the failed execution. We do not provide the details of all the violations represented by the other nodes in the clusters to avoid a useless and tedious discussion of low level implementation details. The nodes in the two top ranked clusters represent violations that are consequences of the violations associated with nodes $V20$ and $V26$ and indicate that Eclipse attempts to manage the `null` values (unexpected interactions mostly related to method `NotificationImpl.getNewValue()`).

6 Validation

This section discusses the experimental data that show how the anomaly graphs produced by BCT help developers diagnose faults. In particular, the validation shows that the anomaly graphs produced with BCT:

1. indicate both the causes of failures and the locations of the related faults,
2. support failure analysis by automatically clustering related anomalies, thus providing a more comprehensive view of the failure causes than state-of-the-art anomaly detection techniques, which do not postprocess the detected anomalies,
3. filter out most false positives, and thus reduce the time spent by developers, who would otherwise waste a lot of time analyzing irrelevant information when using techniques that do not handle false positives.

In the introduction, we observed that BCT is particularly well suited when used to analyze either regression problems or rare field failures experienced after long successful runs. We evaluated BCT with case studies that focus on the analysis of regression and field failures. We start by presenting the case studies considered in the empirical evaluations; we then illustrate the empirical setup; we discuss the empirical data about the ability of BCT to diagnose problems and localize faults; we end by discussing the relevance of integrating heterogeneous model violations and the main threats to the validity of the empirical validation.

6.1 Case Studies

In the evaluation, we use three systems of different sizes and complexities: NanoXML [47], Eclipse [48], and Tomcat [49]. NanoXML is an XML parser of about 8,000 lines of code. Eclipse is a development platform; it is extendible with plugins provided by third parties (off-the-shelf components), and consists of about 17 million lines of code. Tomcat is an application server of about 300,000 lines of code. We analyzed the components of all systems treating them as third party OTS components, thus ignoring both the source code, even when available, and the information about the analyzed components themselves. We used the source code only a posteriori to verify that the feedback provided by BCT corresponds to actual faults.

We monitored the system execution using either Aspectwerkz [29], an aspect-oriented framework [50], or the TPTP probe technology [30]. Both Aspectwerkz and TPTP probes extract runtime data from Java systems without requiring the availability of the source code, and capture the method invocations and the associated attribute values. BCT works only with the intercomponent method invocations and ignores the details of the internal computations.

Table 1 summarizes the case studies and the nature of the test suites executed to collect traces, and provides information about the traces used to generate the interaction models with BCT. Column *Case study identifier* associates case studies with names. The prefix *X2Sql, Nano XML v4 to v5* indicates the investigation of regression faults in Nano XML when used together with Xml2Sql. We considered Nano XML as a black box component and we used BCT to monitor only the interactions between Xml2Sql and Nano XML. The investigated faults have been injected in the context of an upgrade of Nano XML from version 4 to 5. The injected faults are implemented by third parties and are publicly available at the SIR repository [51]. The faults injected for each case study are indicated as the suffix of the case study identifiers. The names of the faults match the names used in the SIR repository.

The prefix *Nano XML v4 to v5* indicates the investigation of regression faults in Nano XML. BCT considers the components in Nano XML as black box components and monitors interactions between these components. We injected the faults of the SIR repository after upgrading Nano XML from version 4 to 5, and refer to the faults by suffixing the identifier either with the SIR case study identifiers or with *all_f* to indicates the injection of all faults.

The *Nano XML v5 to v5, all_f* case study simulates field failures since the faults reported in the SIR repository are injected directly in version 5.

In the case studies of Nano XML with Xml2Sql, BCT analyzes the behaviors at the interfaces between Nano XML and Xml2Sql; while in the set of case studies without Xml2Sql, BCT addresses the faults by monitoring interactions between the components internal to Nano XML. In the case studies with Xml2Sql, we did not consider the fault XER_HD_1 because the fault does not cause the failure of any test case in the Xml2Sql test suite.

The case study *Tomcat 5.5.12 to 5.5.13* consists of the analysis of a regression problem introduced when upgrading from version 5.5.12 to 5.5.13 [52], while the case study *Tomcat 6.0.4* consists of the analysis of a regression fault experienced with version 6.0.4 [53]. BCT analyzes these problems by monitoring the interactions between the main components of Tomcat.

The Eclipse case study *Eclipse, WTP, EMF, GEF, JEM* consists of the analysis of an undocumented regression fault introduced when both updating from Eclipse 3.2.1 with the plugin EMF version 2.2.1 to Eclipse 3.2.2 with EMF 2.2.2 and adding the plugins WTP 1.5.1, GEF 3.2.1, and JEM 1.2.1. The case study *Eclipse EMF 2.2.1, WTP* is the analysis of an undocumented field failure that we experienced with Eclipse 3.2.1 with plugins EMF 2.2.1 and WTP 1.5.1, and is related to the compatibility of the configuration with the Linux operating system. The Eclipse case study *Eclipse, GMF, EMF, OCL* consists of the analysis of a regression problem introduced when updating GMF version 2.0.0M5 and EMF version 2.3.0M5 to GMF 2.0.0M6 and EMF 2.3.0M6 within Eclipse 3.3 with OCL version 1.1.0 [54].

Finally, *ProbeKit, Eclipse(chmod)* indicates the analysis of a fault experienced with the Eclipse TPTP plugin version 4.4.1 [55], and *ProbeKit, Eclipse(EMT64)* indicates the analysis of a fault experienced with the Eclipse TPTP plugin version 4.3 [56]. BCT analyzes these problems by monitoring the interactions between Eclipse and the installed plugins.

We built the BCT models for both NanoXML and Eclipse when executed without the Probekit by executing the test cases that are provided with the applications. The Tomcat application server and the Eclipse Probekit are not distributed with test suites. We designed test cases for the Tomcat Web application manager and the Eclipse instrumentation engine with the category partition method [57].

The data in column *traces* show that BCT can generate models from a relevant number of traces. The average length of the traces is not high because components often produce short interaction sequences. However, several times interactions are very long, up to thousands of events. These data confirm that BCT can process a relevant number of traces that include long as well as compact traces into models by exploiting the regularity in the behavior of software components.

6.2 Empirical Setup

We measured the effectiveness of BCT in both diagnosing problems and locating faults in different ways:

- **Diagnosing problems**: We evaluated the capability of BCT to diagnose problems by measuring its capability to discard irrelevant information, generate focused outputs, and extract all and only valuable information. We evaluated the capability of BCT to discard the irrelevant information that would complicate the interpretation of the output by measuring the number of false positives that BCT filters out automatically. We evaluated the capability of BCT to generate focused outputs by measuring the proportion of true and false positives that developers must inspect before identifying the failure causes. We measured the capability of BCT to extract all and only valuable information from failing executions through the standard measures of precision and recall on the basis of the true and false positives inspected by developers and the false negatives that we manually discovered.
- **Localizing faults**: We evaluated the ability of BCT to localize faults by measuring the distance between the behavioral anomalies detected by BCT and the actual faults that generated the anomalies. We quantify this distance as the number of statements that developers must inspect before reaching the fault location from the statements that produced the anomalies. We compared the accuracy of the results produced by BCT with the Tarantula fault localization technique that represents the state of the art in this field [58].

For each case study, we inspected the analyzed failing executions, and collected both the set of the intercomponent method invocations and the data values exchanged between the components that are related to the failures. We define *failure evidence* as the set of the incorrect intercomponent interactions and the incorrect data values that are exchanged between components as direct or indirect consequence of a revealed fault. We define *strong failure evidence* as the subset of the failure evidences detected at the interfaces of either the faulty component or the faulty integration that caused the failure under investigation. We define *recovery evidence* as the set of the correct intercomponent interactions and the correct data values that are exchanged between components as unique consequence of the failure. This typically happens when systems try to recover from failures, for example recovering from exceptions. Both failure and recovery evidences are relevant to understand the failure causes because they indicate how the system failed and how it tried to recover, respectively. We define the set of *failure cause evidence* as the union of the failure evidence and recovery evidence.

We denote the elements relevant in the BCT analysis as *true positives*, *false negatives*, and *false positives* as follows: *True positives* are the anomalies that are detected by BCT and belong to the failure cause evidence, *false negatives* are the anomalies that are not detected by BCT but belong to the failure cause evidence, *false positives* are the elements that are detected by BCT but do not belong to the failure cause evidence. Our goal is to maximize true positives since they represent actual problems revealed by BCT and minimize false positives since they disturb the information produced by BCT being elements erroneously identified as anomalies.

We say that developers *identify a failure cause* by inspecting the output produced by BCT when the elements

TABLE 2
Summary of the Experiments on the Analysis of Regression Failures

	Anomalies					Connected Components			
	Perc. of	Filtered		Remaining		Total CC	Insp. CC	Insp. Anomalies	
Case Studies	Filtered FP	FP	TP	FP	TP			TP	FP
X2Sql, Nano XML v4 to v5, CR_HD_1	0%	0	0	2	7	2	1	1	0
X2Sql, Nano XML v4 to v5, NV_HD_1	-	0	0	0	**0**	0	0	0	0
X2Sql, Nano XML v4 to v5, SR_HD_1	-	0	0	0	3	1	1	3	0
Nano XML v4 to v5, CR_HD_1	94%	17	0	1	**0**	1	1	0	1
Nano XML v4 to v5, NV_HD_1	95%	21	0	1	**0**	1	1	0	1
Nano XML v4 to v5, SR_HD_1	100%	19	0	0	2	1	1	2	0
Nano XML v4 to v5, XER_HD_1	95%	18	0	1	**0**	1	1	0	1
Nano XML v4 to v5, all_f	63%	10	0	4	6	6	2	6	2
Nano XML v5 to v5, all_f	100%	2	0	0	13	3	3	10	0
Tomcat 5.5.12 to 5.5.13	67%	2	0	1	2	1	1	2	0
Tomcat 6.0.4	100%	2	0	0	2	1	1	2	0
Eclipse, WTP, EMF, GEF, JEM	0%	0	0	1	2	2	2	2	1
Eclipse, EMF 2.2.1, WTP	14%	1	0	6	**0**	2	2	0	6
Eclipse, GMF, EMF, OCL	81%	75	0	18	4	11	2	4	5
ProbeKit, Eclipse(chmod)	99%	90	0	1	3	2	1	3	1
ProbeKit, Eclipse(EMT64)	99%	332	0	2	4	2	1	4	1

(Legend) Column Case Studies *refers to case studies introduced earlier in this section with their identifiers. Column* Perc. of Filtered FP *indicates the percentage of false positives that the filtering heuristic automatically identified and eliminated. Column* Filtered *indicates the number of* FP *and* TP *that the filtering heuristic automatically identified and removed. Column* Remaining *indicates the number of* FP *and* TP *used to generate the connected components presented to the developers. Column* Total CC *indicates the total number of connected components presented to developers. Column* Insp. CC *indicates the number of connected components that developers must inspect to understand the causes of the failures. Column* Insp. Anomalies *indicates the number of relevant* (TP) *and useless* (FP) *anomalies that developers had to inspect before understanding the causes of the failures. Case studies in bold indicate cases where BCT did not provide an interpretation for the causes of the failures. In all such cases, BCT failed because no observable and relevant anomalies were detected, as indicated by the value 0 in the* Remaining TP *column.*

indicated by BCT include at least one element of the strong failure evidence. Intuitively, this indicates that BCT points directly to either faulty components or faulty integration of components, and thus produces information directly related to the causes of the failures and proceed with debugging. For example, in the Tomcat case study discussed in this paper, BCT detects the missing invocation to the static constructor of the `JspFactory` object that is directly responsible for the bootstrapping problem. Thus, BCT includes an element of the strong failure evidence.

6.3 Problem Diagnosis

We empirically evaluated the suitability of BCT to diagnose problems by measuring its ability to extract both valuable and focused information from failing executions. Table 2 reports the experimental data about the false positives that BCT automatically filtered out and about the true and false positives that developers had to inspect to identify the causes of the failures. Table 3 reports the data about precision and recall that quantify the focus of the output of BCT.

As shown in Table 2, the filtering heuristic of BCT is extremely effective in filtering out many false positives without discarding true positives. Column *Perc. of Filtered FP* indicates that the filtering heuristic often identifies more than 90 percent of the false positives, and always misses only a few false positives, while column *Filtered TP* indicates that the heuristic does not erroneously discard any true positive in the considered case studies.

For the regression testing case studies, we reexecuted the available test suites in the new version of the system under analysis; while for case studies on field failures, we reexecuted both the test suite originally used to produce the behavioral models and the inputs that caused the field failures. In both cases, the heuristic removes the model violations that occur in both successful and failing executions. Even if, for field-failures, we reexecuted the same test cases for the same versions of the systems, the heuristic successfully removed some model violations that were specific to the testing sessions executed to produce the models, for instance, violations of I/O models that constrain the current time of the system.

Without efficient filters of false positives, developers waste an enormous amount of time in inspecting many useless model violations. In fact, false positives represent a large portion of the detected model violations when using BCT as well as when using other state-of-the-art techniques.

TABLE 3
Precision and Recall of the BCT Anomaly Detection Capability

Case studies	precision	recall
X2Sql, Nano XML v4 to v5, CR_HD_1	0.67	1
X2Sql, Nano XML v4 to v5, NV_HD_1	-	0
X2Sql, Nano XML v4 to v5, SR_HD_1	1	1
Nano XML v4 to v5, CR_HD_1	0	0
Nano XML v4 to v5, NV_HD_1	0	0
Nano XML v4 to v5, SR_HD_1	1	1
Nano XML v4 to v5, XER_HD_1	0	0
Nano XML v4 to v5, all_f	0.75	0.75
Nano XML v5 to v5, all_f	1	1
Tomcat 5.5.12 to 5.5.13	0.5	0.5
Tomcat 6.0.4	1.00	0.67
Eclipse, WTP, EMF, GEF, JEM	0.67	0.67
Eclipse, EMF 2.2.1, WTP	0	0
Eclipse, GMF, EMF, OCL	0.44	0.67
ProbeKit, Eclipse(chmod)	1	1
ProbeKit, Eclipse(EMT64)	1	0.8

(Legend) Column Case Studies *refers to case studies with their identifiers. Columns* precision *and* recall *specify the value of the precision and recall, respectively. We use the symbol — when the precision cannot be computed due to the absence of both true and false positives.*

Our results confirm the efficacy of the filtering strategies that have been originally introduced in [21], [22] and that we adapted to BCT.

BCT not only filters out many false positives, but also aggregates and prioritizes the results, as discussed in the former section and exemplified in Figs. 10 and 11. We inspected the components after filtering the false positives in the order of priority proposed by BCT, from the largest to the smallest. The effectiveness of aggregation and prioritization is illustrated in the right-hand side of Table 2. Column *Remaining* indicates the number of anomalies after filtering, while columns *Total CC* and *Insp. CC* report the number of components obtained by aggregating the anomalies and the number inspected to finalize the diagnosis, respectively. We can see that developers are given only few components to inspect (11 in the worst case), and that they can diagnose the problem by inspecting even fewer components (three in the worst case), thus confirming the efficacy of BCT.

The same connected component may contain several anomalies that are all inspected by the developers. Columns *Insp. Anomalies TP* and *FP* report the number of true and false positives inspected by developers before identifying the failure causes, respectively. In most cases, developers inspect only true positives. Even when they must inspect some false positives, their number is very small, 1.19 on average and 6 in the worst case. This confirms that the filtering strategy together with the aggregation and prioritization strategies are very effective in focusing on the core problems and removing most of the noise represented by false positives.

The clustering algorithm facilitates the work of the developers in two ways: It groups model violations according to their likely causes, thus focusing the inspection on subsets of related violations, and eliminates some false positives, thus further reducing their noise. The data reported in Table 2 indicate that the clustering strategy has been effective in focusing the attention of the developers in 8 out of 16 cases, and it also reduced the set of anomalies to be inspected in 5 cases. The advantages of clustering is particularly evident in the case of row `Eclipse, GMF, EMF, OCL` of Table 2 where developers find the fault after inspecting 9 model violations grouped in two clusters out of the 22 model violations grouped in 11 clusters.

BCT did not identify valuable information for diagnosing faults in 5 of the 16 case studies, four related to Nano XML and one to Eclipse. In all cases, none of the monitored anomalies were related to the causes of the observed failures. However, in all of these cases developers have to inspect only a few connected components (at the most 2) and few false positives (at the most 6) to realize that BCT is not conclusive. This confirms the efficiency of BCT also when inconclusive.

We examined the case studies in detail to estimate the efficacy of the information produced by BCT with respect to the information usually available to developers, such as the stack trace. In all cases, we found that the information produced by BCT provides more detailed information about the faults than classic information on the stack trace. Here we illustrate the efficacy of the information produced by BCT qualitatively, referring to the Tomcat fault represented by the anomaly graph shown in Fig. 6. As discussed in the previous sections, the anomaly graph generated by BCT points directly to the failure, revealing the presence of a `NullPointerException`, showing its unique relation to the failing execution and indicating the origin of the exception in the postponed execution of an initialization method, which is a direct effect of the fault under analysis. On the other hand, the stack trace indicates only the presence of a `NullPointerException`.

In several case studies, the failure does not generate exceptions or crashes, but only incorrect results. In all of these cases the stack trace can only provide a (sometimes extremely long) list of methods that are active at the time of the failure, without any information about the possible presence of suspicious events in this list (if any), while BCT provides focused information on the anomalies and their relation to the failures.

To further quantify the capability of BCT to extract valuable information, we measure the precision and the recall of the anomaly detection capability of BCT. The precision is defined as the percentage of true positives detected by BCT with respect to the overall number of detected anomalies, that is, $precision = \frac{TP}{TP+FP}$. The recall is defined as the percentage of true positives detected by BCT with respect to the overall number of anomalous behaviors that should have been observed, that is $recall = \frac{TP}{TP+FN}$.

As shown in Table 3, in the successful cases precision and recall are quite high, with an average value of 0.82 for both. This further confirms the good capability of BCT to focus on the relevant information only without accidentally missing important data.

In summary, the results of the experiments confirm our hypothesis. In most of the cases (11 out of 16-69 percent of the examined failures), BCT provided enough information to identify the causes of the failures. In all cases, developers could draw a conclusion by examining a small amount of anomalies: The highest number of inspected anomalies is 10 distributed among three connected components. These excellent results are due to the effectiveness of both the filtering strategy (it often filtered out more than 90 percent of the false positive, with an average of 70 percent) and the aggregation and prioritization mechanisms (the average value of both precision and recall when BCT is successful is 0.82).

6.4 Fault Localization

The main goal of BCT is to generate a set of connected components that explains the causes of failures. In the previous section we provided experimental evidence of the effectiveness of BCT in meeting its main goal. In our experiments, we observed that the code regions corresponding to the anomalies generated by BCT can help developers localizing faults. I/O models are violated by methods called with unexpected parameter values, while interaction models are violated by unexpected method calls. Both unexpected parameter values and method calls may help the developers in localizing the faults that caused the analyzed failures. In this section, we provide experimental results about the ability of BCT to support fault localization.

We measure the ability of BCT to locate faults as the distance between the anomalies revealed by BCT and the

actual faults. The distance represents the effort of inspecting the data to locate the faults. To assess the quality of the results with respect to state-of-the-art techniques, we compared the data produced by BCT with the output produced by the Tarantula fault localization technique [58]. The distance between the information produced by the outputs of a fault localization technique and the actual faults is usually measured as the number of statements that developers must inspect to locate the fault starting from the data provided by the fault localization technique. To avoid biases we conducted the experiments with the case studies that we illustrated earlier in this section and that correspond to faults described in the literature and analyzed by independent parties. We measure the distance between the data provided by either BCT or Tarantula and the actual faults independently from the inspectors, as illustrated below. We referred to the fixes publicly available to identify the fault location of all the known faults, and we located the faults corresponding to the two issues that we discovered in Eclipse by running multiple debugging sessions.

Tarantula produces a ranked list of statements to be inspected in the given order. We measure the inspection effort as the position of the faulty statement in the ranked list. When faulty statements have the same rank of other statements in the list, we consider the average number of statements with the same ranking as the faulty one. For instance, for a faulty statement ranked in the third position together with other two statements, we computed an inspection effort of four.

BCT produces a set of anomalies grouped in connected components ranked by size. Developers inspect the code backwards, starting from the statements that correspond to anomalies. Without additional information developers cannot distinguish between true and false positives in the final output of BCT. Thus, we evaluated the inspection effort assuming that developers inspect anomalies that may correspond to either true or false positives. In actual inspection sessions, we observed that developers can often identify false positives early, and thus we produced conservative results. We assume that developers inspect connected components according to the their ranking, and within each component, they inspect the statements corresponding to anomalies by visiting the components from the root to the leaves according to the edge weight, from the lowest to the highest weight. For each anomaly, developers move backward in the Program Dependency Graph (PDG), as suggested by Cleve and Zeller in [59]. In the presence of multiple equivalent choices, for instance, multiple root nodes or multiple edges with the same weight, we consider the average number of inspected statements as for Tarantula.

We measured the distance between the output produced by either BCT or Tarantula and the fault location as the number of statements that must be inspected before locating the fault, according to the strategies discussed above. When the distance is greater than 50, we classify the output of the technique as useless for locating faults.

Table 4 reports the data of the 16 case studies. The dash symbol indicates that BCT is inconclusive due to the absence of true positives. In these cases, the proximity of anomalies and faults would be purely coincidental. Both

TABLE 4
Number of Statements Inspected with BCT and Tarantula to Reach the Fault Location

Case studies	BCT	Tarantula
X2Sql, Nano XML v4 to v5, CR_HD_1	32	>50
X2Sql, Nano XML v4 to v5, NV_HD_1	-	>50
X2Sql, Nano XML v4 to v5, SR_HD_1	1	26.5
Nano XML v4 to v5, CR_HD_1	-	>50
Nano XML v4 to v5, NV_HD_1	-	12
Nano XML v4 to v5, SR_HD_1	1	47
Nano XML v4 to v5, XER_HD_1	-	>50
Nano XML v4 to v5, all_f	>50	>50
Nano XML v5 to v5, all_f	>50	>50
Tomcat 5.5.12 to 5.5.13	>50	>50
Tomcat 6.0.4	>50	>50
Eclipse, WTP, EMF, GEF, JEM	>50	>50
Eclipse, EMF 2.2.1, WTP	-	>50
Eclipse, GMF, EMF, OCL	>50	>50
ProbeKit, Eclipse(chmod)	1	36
ProbeKit, Eclipse(EMT64)	1	19.5

(Legend) Column Case Studies *refers to case studies with their identifiers. Columns* BCT *and* Tarantula *specify the number of statements inspected by developers before discovering the fault location using BCT and Tarantula, respectively. The symbol — denotes the cases where BCT did not identify any true positive that can support fault localization.*

BCT and Tarantula provide valuable information to locate faults for 5 out of 16 case studies. Note that for all cases where BCT detected at least a true positive, BCT performs better than Tarantula. These results confirm the suitability of BCT to locate faults, but cannot be generalized in terms of relative performances of the two techniques since they depend on both the size of the test suites and the nature of the analyzed faults. The available test suites are quite small compared to the amount of functionalities of the analyzed systems, and Tarantula suffers more than BCT from limited code sampling. BCT concentrates on the behavioral differences that the systems exhibit when the same test suites are reexecuted to identify the causes of failures. The quality of the results depends on the homogeneity of the portions of the execution space sampled by the same test suites and not on the coverage of the whole execution space, while Tarantula works well on a good sampling of the whole execution space. Moreover, the analyzed failures depend on faulty integration of components, which is the target of BCT, while Tarantula works better on faults at unit level.

The distance between the outputs produced by BCT and the corresponding fault locations varies significantly from case to case: from 1 to over 50. This high variability depends on the information provided by BCT. BCT monitors component interactions, and thus identifies only code locations at the component boundaries, i.e., method calls. The distance between anomalies and faults depends on the PDG distance from the actual fault location and the nearest method call from the faulty component to another component of the system. The distance depends on the nature of the components: It can be large for complex components, like the ones that we analyzed in the Tomcat and the Eclipse case studies, while it is often small for simple components, like the ones that we analyzed in the Nano XML case studies.

We conclude this section with a qualitative observation. When techniques based on code coverage fail to indicate statements close to the fault locations, developers must work independently from the technique. On the contrary,

TABLE 5
Distribution of the Anomalies between I/O and Interaction Anomalies

Case Studies	Perc. TP IO	Perc. TP Interactions
X2Sql, Nano XML v4 to v5, CR_HD_1	29%	71%
X2Sql, Nano XML v4 to v5, SR_HD_1	67%	33%
Nano XML v4 to v5, SR_HD_1	50%	50%
Nano XML v4 to v5, all_f	33%	67%
Nano XML v5 to v5, all_f	23%	77%
Tomcat 5.5.12 to 5.5.13	0%	100%
Tomcat 6.0.4	50%	50%
Eclipse, WTP, EMF, GEF, JEM	0%	100%
Eclipse, GMF, EMF, OCL	25%	75%
ProbeKit, Eclipse(chmod)	67%	33%
ProbeKit, Eclipse(EMT64)	85%	25%

(Legend) Column Case Studies *refers to case studies with their identifiers. Column* Perc. TP *specifies the distribution of the true positives between I/O and interaction anomalies.*

BCT provides valuable information to identify the causes of failures independently from the distance between the anomalies and the fault location. The structure of anomaly graphs allows developers to navigate between different code locations according to their understanding of the causes of failures and their knowledge of the application, following their intuition rather than the PDG. For example, when examining a violation that traces back to a non-initialized variable, the developers would likely go directly to the function that initializes the components, rather than follow the PDG. This characteristic of the data provided by BCT can significantly reduce the effort required to localize faults, but is hard to measure objectively, and thus is not quantified in this paper.

6.5 Heterogeneous Model Violations

BCT produces I/O and interaction models. Table 5 provides data about the contribution of the two kinds of models to the diagnosis of the faults. For each case study, the table shows the relative amount of true positives that correspond to I/O and interaction model violations, respectively (column *perc. TP*). Although violations of interaction models prevail, the table shows that violations of both classes of models contribute in diagnosing the faults: Both classes of models produced relevant anomalies in 9 out of 11 successful cases, thus suggesting the usefulness of combining different models. Focusing on a single kind of anomaly would strongly limit the amount of information available to analyze failing executions.

6.6 Threats to Validity

We conducted all of the experiments on failures experienced with real systems during either regression testing or field executions. In all cases, we could count on a thorough set of executions to build our dynamic models. The experimental data reported in this paper cannot be generalized to cases for which there are not many executions. BCT is likely to be less effective when dealing with systems that have not been thoroughly executed, but we believe that the analysis results of BCT could help developers also when dealing with cases not thoroughly executed.

We report the results of a relatively small set of experiments: a total of 16 cases, out of which 9 cases contain faults seeded by third parties and 7 cases contain real faults. The results are good, but not statistically relevant yet. We executed a relatively small set of experiments because of the cost of setting up the experiments. The core parts of the experiments are not expensive: The overhead of running BCT is low with respect to the whole process, and the cost of locating faults with BCT is lower than with classic inspection techniques, but setting up the experiments and manually inspecting the failing executions to identify incorrect interactions and establish the nature of component interactions can be extremely time consuming. In particular, to generate dynamic models, to produce the refined anomaly graph, and to examine the results to understand if the graphs provide enough information to diagnose faults, we must generate test suites (if not already available), deploy the systems, execute the test cases, reproduce the failures, and manually inspect the failing executions step by step. All activities, and especially the final inspection, are extremely demanding. This set of experiments, defined with real systems that present both real faults and faults seeded by third-parties, reduces the bias that may be induced by a relatively small number of experiments. The quantitative results may be imprecise, but the results obtained with the case studies give us confidence on the effectiveness of the technique.

We conducted all experiments on mid-sized systems. The complexity and the cost of setting up and manually inspecting the failing executions did not allow us to extend the experiments to large systems. However, the complexity of the experiments themselves and the cost of running BCT do not increase with the size of the system. BCT generates an I/O model and an interaction model for each interface method of each monitored component. The size of each model does not depend on the size of the system, but rather on the amount of interactions between the monitored component and the rest of the system, which does not grow significantly for well engineered systems. Thus, with systems of increasing size, we experience a growing number of models, but not a growing complexity of the single models. The cost of generating anomaly graphs depends on the portion of the dynamic call tree that corresponds to the failed execution, which again does not depend on the size of the model. The technique for refining anomaly graphs depends on the anomaly graph and not on the size of the system. Thus, we have no reason to doubt that the results obtained on mid-sized systems apply to large sized systems as well.

Finally, we identify true positives, false positives, and false negatives by manually inspecting the failing executions. This process may be imprecise, especially when inspecting complex systems with limited knowledge on their design. We addressed this risk by having the failing executions inspected carefully and multiple times by different inspectors.

7 Related Work

There are many approaches to automatically identify failure causes and to diagnose faults. Here, we focus on techniques that work without precise and complete specifications, and that classify as techniques for locating faults by analyzing the executed code, techniques for diagnosing faults by

```
58: public void lifecycleEvent(LifecycleEvent event) {
59:
60: if (Lifecycle.INIT_EVENT.equals(event.getType())) {
61:   try {
62:     // Set JSP factory
63:     this.getClass().getClassLoader().loadClass
64:       ("org.apache.jasper.compiler.JspRuntimeContext");
65:   } catch (Throwable t) {
66:     // Should not occur, obviously
67:     log.warn("Couldn't initialize Jasper", t);
68:   }
```

Fig. 12. An excerpt of JasperListener.java from lines 58 to 68. The faulty statement that causes the failure in Tomcat 6.0.4 is in lines 63 and 64.

detecting execution anomalies, and techniques for diagnosing faults that combine static and dynamic analysis.

7.1 Locating Faults by Analyzing the Executed Code

Techniques for locating faults based on the analysis of the executed code elements assume that elements frequently executed in failed executions and rarely executed in valid executions are more likely faulty than elements executed both in success and failed executions (or only in successful executions) [10], [11], [12], [13], [14], [15]. These techniques provide useful information to localize faults, but little support for interpreting the problem under investigation. In fact, when successful, the identified fault locations indicate the code that should be modified to fix the faults, but rarely provide information to diagnose the causes of the failures and to understand the nature of the faults. On the contrary, BCT provides coarser information about the statements to be fixed, but useful information about the causes of the failure and the required fix.

We illustrate the differences between these techniques and BCT by discussing how *Tarantula* works with the Tomcat case study presented in the former section. *Tarantula* can correctly identify the faulty statement at lines 63 and 64, as shown in Fig. 12, but not the cause of the failure and the required fix. In particular, pointing at the statement does not provide any evidence that the fault is an uninitialized `JSPFactory` object, and that this is due to an incorrect implementation of static class constructors.

BCT is less precise than Tarantula since it localizes faults at the granularity of methods rather than code blocks, but the refined anomaly graphs produced by BCT point both to a violation of an interaction model that indicates that the failed execution does not include a call to a static constructor and a violation of an I/O model that indicates an unexpected `null` value as the return value of `getDefaultFactory()` (see Fig. 6). This suggests the fault fixing that consists of forcing the call to the class constructor before loading the web applications.

Techniques based on the analysis of the executed code elements work well with faults localized in few code elements, whose execution frequently leads to failures. They are much less effective with failures caused by differences in the inputs, failures related to missing events, and failures caused by multiple faults rather than failures depending on the executed paths, since there is no clear relation between the executed code and the failures themselves. BCT complements these techniques by addressing faults that cause anomalous component interactions, like failures caused by specific input values, by specific sequences of method calls and by rare combinations of events.

Finally, fault localization techniques usually assume access to the source code, while BCT does not access the source code and does not suffer when it is not available, for instance, when OTS components are part of the system under analysis.

7.2 Diagnosing Faults by Detecting Execution Anomalies

The techniques that diagnose faults by detecting execution anomalies compare failed executions with models of successful executions. Here, we review the techniques that work with models dynamically inferred from program executions that can work with systems released without precise and complete specifications.

These techniques provide useful data about the causes of the failures that they can analyze, but limited information about the locations of the corresponding faults [16], [17], [18], [60], [61]. Moreover, they usually refer to a single type of model and thus target only failures that can be detected with the adopted model. For example, Wasylkowski et al. infer finite state automata to detect anomalies caused by object interactions [17], while Raz et al. infer I/O models to detect anomalies caused by data exchanged with online services [18]. Since anomalous behaviors are not necessarily faulty behaviors, these techniques generate many false positives that heavily impact the effort required to identify the relevant data. BCT combines finite state automata and I/O models, thus targeting a larger set of failures, and integrates filtering and clustering techniques that can both eliminate many false positives and deal with multiple faults. Both the techniques for filtering false positives and clustering anomalies can be generalized to many different models.

Other techniques that dynamically derive models do not suggest how to use the generated models to identify anomalies, even if they have the potential. For example, Daikon derives Boolean expressions to describe relations between program variables [40]. Many finite state automata inference algorithms identify constraints on the ordering of events [37], [38], [41], [42]. The technique proposed by Henckel et al. derives algebraic specifications that describe the behavior of container classes [62]. BCT complements these technologies by proposing a technique that use models to describe the likely causes of failures and to diagnose the corresponding faults while filtering false positives.

7.3 Diagnosing Faults by Combining Static and Dynamic Analysis

Some approaches combine dynamic and static analysis to diagnose faults. Two important representatives are DSD-Crasher [63] and the technique by McCamant and Ernst [64].

DSD-Crasher effectively combines the Daikon invariants with the static exploration of possible program behaviors of Check 'n' Crash [65] to identify executions that likely result in program failures. DSD-Crasher also exploits Check 'n' Crash capabilities to verify whether the identified executions effectively result in crashes, thus pruning false positives. To eliminate false positives, DSD-Crasher generates concrete executions that can reproduce the failures caused by the identified faults. DSD-Crasher can only reveal faults that result in system crashes, requires the

source code and does not scale well to large systems, because of the limitations of the integrated static analysis techniques [63]. While BCT addresses a larger set of fault types, does not require the source code, and scales to large systems, but is less effective than DSD-Crasher in eliminating false positives.

The technique proposed by McCamant and Ernst automatically infers Daikon invariants that describe the preconditions of component interfaces and statically analyzes the generated invariants to verify the compatibility between component upgrades. This technique uses Daikon invariants to identify possible incompatibilities a priori, while BCT uses both Daikon invariants and FSA to dynamically detect anomalies and to identify likely causes of failures and locations of faults. The two techniques differ also in scope: BCT focuses on the behavior of the whole system, while the technique of McCamant and Ernst addresses the behavior of single components.

Several other techniques combine static and dynamic analysis to augment test suites [66], [67], [68], [69], [70]. These techniques do not aim to identify failure causes, but to enhance the possibility of revealing failures and thus have great potentialities, if integrated within BCT.

Finally, a combination of static and dynamic analysis has been investigated in the ClearView technique to automatically generate security patches [71]. ClearView shares some ideas with BCT, such as the use of dynamic analysis to identify behavioral anomalies and the heuristic correlation of the anomalies. However, ClearView is narrowed to a small subset of the Daikon invariants and focuses on the generation of patches for security issues, while BCT uses a heterogeneous and wider set of anomalies and targets the different goal of automatically identifying failure causes and fault locations.

8 Conclusion

In this paper, we presented a technique called BCT that helps developers to diagnose failures and to locate the related faults. Differently from other works to automatically diagnose faults, BCT does not require access to either the specifications or the source code of the system components. Thus, it applies well to systems built from OTS components that are provided with partial specifications and no access to source code. BCT locates failure causes and diagnoses faults by comparing failing executions with models of successful executions that are dynamically generated by monitoring system runs, and includes an effective algorithm to generate hybrid models of component interactions that capture both execution sequences and constraints on data exchanged among components during the computation. BCT also includes an algorithm to weight the relation among model violations based on a simple heuristic, and includes techniques to automatically filter out false positives and cluster model violations related to the same fault. In this way, BCT extracts focused information that helps developers identify the causes of failures and diagnose faults, significantly reducing the time wasted in examining useless data.

We evaluated BCT in two relevant scenarios—regression testing failures and rare failures experienced after many successful runs. In both cases, experiments with real cases show that BCT can effectively locate the causes of failures and diagnose the related faults.

We are currently investigating how to incrementally maintain models while software evolves. Our current research work focuses on how to identify the changed behaviors, how to add and remove behaviors from models without reexecuting the entire test suites and how to extend models by learning from false positives.

Acknowledgments

This work has been partly supported by the European Community under the Information Society Technologies (IST) program of the sixth FP for RTD—project SHADOWS contract IST-035157.

References

[1] SAP "Crystal Report," http://www.sap.com/solutions/sapbusinessobjects/sme/reporting/crystalreports/, 2009.
[2] JasperForge "Jasper Report," http://jasperforge.org/, 2009.
[3] Eclipse Community "Eclipse Plugin Central," http://www.eclipseplugincentral.com, 2009.
[4] M. Morisio, C.B. Seaman, A.T. Parra, V.R. Basili, S.E. Kraft, and S.E. Condon, "Investigating and Improving a COTS-Based Software Development," *Proc. 22nd Int'l Conf. Software Eng.*, pp. 32-41, 2000.
[5] N.G. Leveson, "The Role of Software in Spacecraft Accidents," *AIAA J. Spacecraft and Rockets*, vol. 41, no. 4, pp. 564-575, 2004.
[6] D. Hovemeyer and W. Pugh, "Finding Bugs Is Easy," *Proc. 19th Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 92-106, 2004.
[7] N. Rutar, C.B. Almazan, and J.S. Foster, "A Comparison of Bug Finding Tools for Java," *Proc. IEEE 15th Int'l Symp. Software Reliability Eng.*, pp. 245-256, 2004.
[8] M. Zitser, R. Lippmann, and T. Leek, "Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code," *Proc. 12th Int'l Symp. Foundations of Software Eng.*, pp. 97-106, 2004.
[9] S. Hissam and D. Carney, "Isolating Faults in Complex COTS-Based Systems," white paper, Carnegie Mellon Software Eng. Inst., 1998.
[10] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*, Morgan Kaufman. 2005.
[11] M. Renieris and S. Reiss, "Fault Localization with Nearest Neighbor Queries," *Proc. IEEE 18th Int'l Conf. Automated Software Eng.*, pp. 30-39, 2003.
[12] L. Briand, Y. Labiche, and X. Liu, "Using Machine Learning to Support Debugging with Tarantula," *Proc. IEEE 18th Int'l Symp. Software Reliability Eng.*, pp. 137-146, 2007.
[13] W. Masri, A. Podgurski, and D. Leon, "An Empirical Study of Test Case Filtering Techniques Based on Exercising Information Flows," *IEEE Trans. Software Eng.*, vol. 33, no. 7, pp. 454-477, July 2007.
[14] J. Jones, M. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," *Proc. 24th Int'l Conf. Software Eng.*, pp. 467-477, 2002.
[15] A. Zeller, "Isolating Cause-Effect Chains from Computer Programs," *Proc. 10th Symp. Foundations of Software Eng.*, pp. 1-10, 2002.
[16] S. Hangal and M.S. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," *Proc. 24th Int'l Conf. Software Eng.*, pp. 291-301, 2002.
[17] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting Object Usage Anomalies," *Proc. Joint Meeting European Software Eng. Conf. and Symp. Foundations of Software Eng.*, 2007.
[18] O. Raz, P. Koopman, and M. Shaw, "Semantic Anomaly Detection in Online Data Sources," *Proc. 24th Int'l Conf. Software Eng.*, pp. 302-312, 2002.
[19] L. Mariani and M. Pezzè, "Dynamic Detection of COTS Components Incompatibility," *IEEE Software*, vol. 24, no. 5, pp. 76-85, Sept./Oct. 2007.

[20] L. Mariani and M. Pezzè, "Behavior Capture and Test: Automated Analysis of Component Integration," *Proc. IEEE 10th Int'l Conf. Eng. Complex Computer Systems,* pp. 292-301, 2005.
[21] C. Liu and J. Han, "Failure Proximity: A Fault Localization-Based Approach," *Proc. 14th Int'l Symp. Foundations of Software Eng.,* pp. 101-112, 2006.
[22] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, and M.I. Jordan, "Scalable Statistical Bug Isolation," *Proc. Conf. Programming Language Design and Implementation,* pp. 15-26, 2005.
[23] Eclipse "Bug Reports for Eclipse Project," https://bugs.eclipse.org/bugs/, 2009.
[24] S. Eldh, S. Punnekkat, H. Hansson, and P. Jönsson, "Component Testing Is Not Enough—A Study of Software Faults in Telecom Middleware," *Proc. 19th IFIP Int'l Conf. Testing of Communicating Systems,* 2007.
[25] M.J.P. van der Meulen, P. Bishop, and R. Villa, "An Exploration of Software Faults and Failure Behaviour in a Large Population of Programs," *Proc. IEEE 15th Int'l Symp. Software Reliability Eng.,* pp. 101-112, 2004.
[26] S. Chandra and P.M. Chen, "Whither Generic Recovery from Application Faults? A Fault Study Using Open-Source Software," *Proc. IEEE Int'l Conf. Dependable Systems and Networks/Symp. Fault-Tolerant Computing,* pp. 97-106, 2000.
[27] E. Weyuker, "Testing Component-Based Software: A Cautionary Tale," *IEEE Software,* vol. 15, no. 5, pp. 54-59, Sept./Oct. 1998.
[28] IEEE Standards Board "IEEE Standard Glossary of Software Engineering Terminology," Technical Report Std 610.12-1990, IEEE, 1990.
[29] Aspectwerkz, http://aspectwerkz.codehaus.org/, 2009.
[30] IBM, "Eclipse Test & Performance Tools Platform," http://www.eclipse.org/tptp/, 2009.
[31] Sun, "Java Reflection," http://java.sun.com/javase/6/docs/technotes/guides/reflection/, 2009.
[32] J.H. Perkins and M.D. Ernst, "Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants," *Proc. 12th ACM SIGSOFT,* pp. 23-32, 2004.
[33] J.A. Clause and A. Orso, "A Technique for Enabling and Supporting Debugging of Field Failures," *Proc. IEEE 29th Int'l Conf. Software Eng.,* pp. 261-270, 2007.
[34] J. Oncina and P. Garcia, "Inferring Regular Languages in Polynomial Update Time," *Pattern Recognition and Image Analysis,* N. P. de la Blanca, A. Sanfeliu, and E.Vidal, eds., pp. 49-61, World Scientific. 1992.
[35] S. Porat and J. Feldman, "Learning Automata from Ordered Examples," *J. Machine Learning,* vol. 7, pp. 109-138, 1991.
[36] O. Cicchello and S.C. Kremer, "Inducing Grammars from Sparse Data Sets: A Survey of Algorithms and Results," *J. Machine Learning Research,* vol. 4, pp. 603-632, 2003.
[37] A. Biermann and J. Feldman, "On the Synthesis of Finite State Machines from Samples of Their Behavior," *IEEE Trans. Computers,* vol. 21, no. 6, pp. 592-597, June 1972.
[38] J. Cook and A. Wolf, "Discovering Models of Software Processes from Event-Based Data," *ACM Trans. Software Eng. and Methodology,* vol. 7, no. 3, pp. 215-249, 1998.
[39] S.P. Reiss and M. Renieris, "Encoding Program Executions," *Proc. IEEE 23rd Int'l Conf. Software Eng.,* pp. 221-230, 2001.
[40] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Trans. Software Eng.,* vol. 27, no. 2, pp. 99-123, Feb. 2001.
[41] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic Generation of Software Behavioral Models," *Proc. 30th Int'l Conf. Software Eng.,* 2008.
[42] R. Parekh, C. Nichitiu, and V. Honavar, "A Polynomial Time Incremental Algorithm for Learning DFA," *Proc. Fourth Int'l Colloquium Grammatical Inference,* 1998.
[43] L. Mariani, F. Pastore, and M. Pezzè, "kBehavior: Algorithms and Complexity," LTA Technical Report lta-2010-01, Univ. of Milano Bicocca, http://www.lta.disco.unimib.it/lta/teamPublications.php, 2010.
[44] G. Ammons, T. Ball, and J.R. Larus, "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling," *Proc. Conf. Programming Language Design and Implementation,* pp. 85-96, 1997.
[45] A. Gordon, *Classification,* second ed. Chapman and Hall/CRC, 1999.
[46] W.-K. Chen, *Graph Theory and Its Engineering Applications.* World Scientific Publishing Company, 1997.
[47] M.D. Scheemaecker, "Nano XML," http://nanoxml.cyberelf.be/, 2009.
[48] Eclipse Foundation, "Eclipse," http://www.eclipse.org/, 2009.
[49] The Apache Software Foundation, "Tomcat," http://tomcat.apache.org/, 2009.
[50] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *Proc. 11th European Conf. Object Oriented Programming,* pp. 220-242, 1997.
[51] Galileo Research Group, "Software-Artifact Infrastructure Repository (SIR)," http://esquared.unl.edu/sir, 2009.
[52] Eclipse Software Foundation, "Tomcat Bug 41939," https://issues.apache.org/bugzilla/show_bug.cgi?id=41939, 2009.
[53] Apache Software Foundation, "Tomcat Bug 40820," https://issues.apache.org/bugzilla/show_bug.cgi?id=40820, 2009.
[54] Eclipse, "Eclipse Bug 181288," https://bugs.eclipse.org/bugs/show_bug.cgi?id=181288, 2009.
[55] Eclipse, "Eclipse Bug 221738," https://bugs.eclipse.org/bugs/show_bug.cgi?id=221738, 2009.
[56] Eclipse, "Eclipse Bug 156532," https://bugs.eclipse.org/bugs/show_bug.cgi?id=156532, 2009.
[57] T.J. Ostrand and M.J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," *Comm. ACM,* vol. 31, no. 6, pp. 676-686, June 1988.
[58] J. Jones and M.J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," *Proc. Int'l Conf. Automated Software Eng.,* pp. 273-282, 2005.
[59] H. Cleve and A. Zeller, "Locating Causes of Program Failures," *Proc. 27th Int'l Conf. Software Eng.,* pp. 342-351, 2005.
[60] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining Temporal API Rules from Imperfect Traces," *Proc. 28th Int'l Conf. Software Eng.,* pp. 282-291, 2006.
[61] C. Yilmaz, A. Paradkar, and C. Williams, "Time Will Tell: Fault Localization Using Time Spectra," *Proc. 30th Int'l Conf. Software Eng.,* pp. 81-90, 2008.
[62] J. Henkel, C. Reichenbach, and A. Diwan, "Discovering Documentation for Java Container Classes," *IEEE Trans. Software Eng.,* vol. 33, no. 8, pp. 526-543, Aug. 2007.
[63] C. Csallner, Y. Smaragdakis, and T. Xie, "DSD-Crasher: A Hybrid Analysis Tool for Bug Finding," *ACM Trans. Software Eng. and Methodologies,* vol. 17, no. 2, pp. 1-37, 2008.
[64] S. McCamant and M.D. Ernst, "Predicting Problems Caused by Component Upgrades," *Proc. Ninth European Software Eng. Conf. Held Jointly with 11th Int'l Symp. Foundations of Software Eng.,* pp. 287-296, 2003.
[65] C. Csallner and Y. Smaragdakis, "Check 'n' Crash: Combining Static Checking and Testing," *Proc. 27th Int'l Conf. Software Eng.,* pp. 422-431, 2005.
[66] T. Xie and D. Notkin, "Tool-Assisted Unit Test Selection Based on Operational Violations," *Proc. IEEE 18th Int'l Conf. Automated Software Eng.,* pp. 40-48, 2003.
[67] M. Harder, J. Mellen, and M.D. Ernst, "Improving Test Suites via Operational Abstraction," *Proc. IEEE 25th Int'l Conf. Software Eng.,* pp. 60-71, 2003.
[68] P. Godefroid, "Compositional Dynamic Test Generation," *Proc. 34th Symp. Principles of Programming Languages,* pp. 47-54, 2007.
[69] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," *Proc. Conf. Programming Language Design and Implementation,* pp. 213-223, 2005.
[70] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic unit Testing Engine for C," *Proc. 13th Int'l Symp. Foundations of Software Eng.,* pp. 263-272, 2005.
[71] J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. Ernst, and M. Rinard, "Automatically Patching Errors in Deployed Software," *Proc. 22nd Symp. Operating Systems Principles,* pp. 87-102, 2009.

Leonardo Mariani received the PhD degree in 2005 from the University of Milano Bicocca. He is a researcher at the University of Milano Bicocca. He spent three months as a visiting researcher at the University of Paderborn. His main research area is software engineering, with specific focus on the following topics: testing and analysis of component-based systems, dynamic analysis, design and development of self-healing solutions, and test and analysis of service-based applications. He served as a program committee member for several conferences related to software engineering and autonomic computing; and chaired several workshops and tracks, such as the ISSRE '10 Fast Abstract track, the ARAMIS '08 workshop, and the GM-VMT '11 workshop. He has worked on several research projects, such as the STREP '06 "SHADOWS" project, which focuses on the definition of a model-based approach to develop self-healing systems, and the STREP '10 "PINCETTE" project, which focuses on the definition of techniques that integrate static and dynamic analysis for the quality assurance of evolving systems. He is a member of the IEEE and the IEEE Computer Society.

Fabrizio Pastore received the MS and PhD degrees in informatics from the University of Milano Bicocca. He is a postdoctoral researcher at the University of Lugano. His research interests regard software engineering and software quality assurance: fault localization through dynamic analysis, regression testing, and development of self-healing systems. While working on his PhD, he collaborated with different research partners for the realization of an automated fault diagnosis and healing framework developed in the context of the EU STREP '06 "SHADOWS" project. He is a member of the IEEE and the IEEE Computer Society.

Mauro Pezzè received the Laurea degree in computer science from the University of Pisa, Italy, in 1984 and the doctorate degree in computer science from the Politecnico di Milano, Italy, in 1989. He is a professor of computer science at the University of Milano Bicocca and the University of Lugano. His general research interests are in the areas of software testing and analysis, autonomic computing, self-healing software systems, service-base applications, and service level agreement protection. Prior to joining the University of Milan Bicocca and the University of Lugano as full professor, he was an assistant and an associate professor at the Politecnico di Milano, and a visiting researcher at the University of Edinburgh and the University of California, Irvine. He is an associate editor of the *ACM Transactions on Software Engineering and Methodology* and member of the steering committee of the ACM International Conference on Software Testing and Analysis (ISSTA) and the International Conference on Software Engineering (ICSE). He is coauthor of the book *Software Testing and Analysis, Process, Principles and Techniques* (John Wiley, 2008) and he is the author or coauthor of more than 80 refereed journal and conference papers. He is a senior member of the IEEE and a member of the IEEE Computer Society.