Dynamic Analysis for Integration Faults Localization

Leonardo Mariani and Fabrizio Pastore and Mauro Pezzè*
University of Milano Bicocca
Department of Informatics, Systems and Communication
viale Sarca 336, 20126 Milano, Italy
{mariani, pastore, pezze}@disco.unimib.it

Abstract

Software developers usually integrate third party components to build systems providing distinct functionalities like graph drawing, or persistence capabilities. Developers often use *grey-box* components: software modules they do not know in details because provided without source code or incomplete specifications or both. Lack of source code and specifications makes the integration of such modules difficult and often causes faults that lead to critical failures if not detected at testing time.

Lack of such informations complicates faults localization too. Existing static analysis and debugging techniques rely on source code or specifications, for this reason their applicability is often limited when grey-box components are used. Dynamic analysis techniques do not need such information: they monitor components interfaces, thus being applicable even when this information is missing. Dynamic analysis approaches identify violations of models inferred from data recorded during monitored executions. Unfortunately these techniques suffers from limitations too: they are capable of identifying only specific kinds of faults, their results are often affected by false positives, and they present scalability issues depending on the huge amount of data collected during training or on the identification of many violations during debugging.

This paper presents *Behaviour Capture and Test (BCT)*, a dynamic analysis technique that overcomes the limitations of the existing dynamic analysis techniques. BCT uses different kinds of models to localize different types of faults, it prunes false positives, it incrementally builds models to save disk space and guides developers when many models violations are identified. Successful results obtained when applying the technique on injected and real faults are reported in the paper: the considered case studies regard both regression faults and faults depending on rare events sequences not stressed at testing time.

*Mauro Pezzè is also professor at the University of Lugano, Faculty of Informatics, via Buffi, 13 6900 Lugano (Switzerland).

1 Introduction

Software systems often include *grey-box* components, that is components that do not expose all their internal details, such as components that are available with poor specifications and only partial view of the internal details, or COTS components that are provided without source code and with incomplete specifications. For example, many vendors offer COTS components that implement CAD drawing features [5], and several web sites offer plug-ins that extend application capabilities [12], commonly offered without source code and with informal (and incomplete) specifications.

Lack of access to source code and incomplete specifications harm the integration of *grey-box* components and lead to new and unpredicted integration faults. For instance, several investigations of aerospace problems indicate integrations faults and incomplete specifications as major sources of critical software failures in aerospace missions [29, 24]. Static analysis techniques can identify and remove some faults, but require access to source code, need formal specifications to investigate many classes of faults, and generate many false positives that reduce their practical applicability [21, 37, 52].

Limited availability of source code and specifications complicates debugging as well. Hissam and Carney discuss the problems that arise when debugging software with partial access to the source code [20]. State of art debugging techniques propose different heuristics to identify fault locations by comparing the code executed in failing and successful executions [51, 36, 4, 28, 22, 50]. Unfortunately, they cannot be applied without access to source code.

Dynamic analysis can produce useful information for identifying and localizing faults even in presence of *grey-box* components. Dynamic analysis monitors system executions by observing interactions at the component interface level, derives models of the expected behavior from the observed events, and marks model violations as sympthoms of faults [19, 48].[1]

State-of-art techniques provide good evidence that dynamic analysis produces useful information to dected faults in systems that include *grey-box* components, but suffer from limitations that hinder their practical applicability:

- Most techniques focus on some aspects, ignoring others that are important in component interactions. For example, Raz, Koopman, and Shaw focus on the data involved in the communication among components, but ignore the interaction order [34], while Wasylkowski, Zeller, and Lindig focus on the interaction order but do not consider the data involved in the communication [48].
- Most model generation techniques store huge amount of data, with obvious consequences on scalability [16, 3].
- Most techniques produce many false positive that are difficult to prune [34].
- None of the currently available techniques can relate multiple model violations that depend on the same fault, thus providing test engineers with information hard to filter and interpret.

[1] In this paper, we use the term *anomaly* to indicate executions that violate dynamic models.

This paper proposes a methodology, called *BCT (Behavior Capture and Test)*, to localize faults and diagnose fault causes. BCT is based on incremental dynamic analysis techniques that extract useful information without expensive storage consumption. This paper advances the state of the art in automatic fault localization by:

- introducing a methodology that combines dynamic analysis and model-based monitoring to provide extensive descriptions of possible fault locations and fault causes in terms of both components and operations likely responsible for failures (location), and structured set of interactions and data values likely related to failures (cause).
- combining classic dynamic analysis techniques (Daikon) with incremental finite state generation techniques (k-behavior) to produce dynamic models that capture complementary aspects of component interactions, and are built incrementally without expensive storage consumption.
- proposing a technique that prunes false positives by compares failing and successful executions
- defining a method to extract information about likely causes of failures by automatically ranking and relating the detected anomalies
- presenting a set of case studies that indicate the effectiveness of the proposed methodology

The effectiveness of BCT depends on the quality of the dynamic model extracted from the program, which is affected by the quality of the test suites and the samples of the input space used to derive the dynamic models: when the test cases sample well the execution space, as in the cases of regression test suites, and of rare field failures experienced after long positive runs, *BCT* works particularly well. The data provided by the test cases reported in this paper show that *BCT* is indeed effective in both cases:

- to detect regression faults that can be experienced when systems, like Eclipse, are extended with components and plug-in available in many versions [10];
- to detect field faults caused by integration problems and rare event sequences that pass unit testing [15, 47, 6, 49].

This paper is structured as follow. Section 2 introduces the BCT technique. Section 3 presents model synthesizers. Section 4 describes techniques to prune false positives and effectively present relevant information to testers. Section 5 presents the application of BCT to several case studies. Finally, Section 6 concludes the paper and outlines future work.

2 Behavior Capture and Test

The *BCT (Behavior Capture and Test)* methodology proposed in this paper is based on two main steps: the construction of detailed (partial) models of the correct behavior of component interactions, and the analysis of such models to locate faults and diagnose fault causes.

BCT *constructs detailed models* of the interactions between software components by means of dynamic analysis. To extract such models, BCT monitors the interactions at the component interface level, without accessing the implementation details of the components, and thus BCT can analyze components provided without access to the source code. BCT can rely on different technologies, like aspect oriented programming [23], or platform probes, such as the ones available within the TPTP platform [13]. Both technologies extract runtime data from programs without requiring availability of source code, and both capture method invocations, in the form of 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 monitoring technologies with a graph-traversal algorithm, called *object flattening*, which navigates the structure of the object graph reachable through captured references and records the extracted information. Object flattening accesses object attributes through reflection [40], visits the object graph with a bread-first visiting strategy, and includes ad-hoc strategies to detect and avoid cycles and multiple extraction of the same object attributes.

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

To locate faults and their causes, we need a model that well approximates the correct behavior of the software systems. BCT builds such models by collecting data in settings where the software is executed thoroughly and successfully. A natural scenario to produce such models is system and acceptance testing, because test cases well exercise the software system, and test oracles indicate whether the test cases passed or not.

BCT incrementally infers models that summarize and generalize the observed behaviors. We generate 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, the I/O model `item.quantity>0` associated with method `Cart.add(Item item)` specifies that the attribute `quantity` of parameter `item` passed to a method `add` implemented by a `Cart` component assumed only positive values in the monitored executions. We generate I/O models with Daikon, which can process data incrementally [32].

Interaction models are Finite State Automata (FSA) that are associated with the methods in the component interfaces, and that summarize the inter-component invocations that are caused by the method execution. For example, the FSA shown in Figure 1 indicates the sequences of method invocations that can be observed when method `order(Cart cart)` is executed to create a new order: 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()`). We discuss in details the incremental inference engine kBehavior in Section 3.

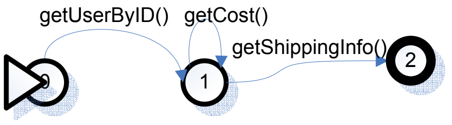


Figure 1: The FSA associated with method `order(Cart cart)`.

BCT *locates faults* and diagnose fault causes by comparing faulty executions, that is executions that lead to failures, with I/O and interaction modes built during correct executions. BCT identifies candidate faults as the events that violate the models built during correct executions, filters false positives by exploiting some heuristics, and clusters the resulting events to describe possible fault causes. We discuss in details heuristics to filter false positives, and techniques to cluster related violations in Section 4.

We compare faulty executions with I/O and interaction models by re-executing the software after a failure occurrence. We can easily reproduce failures that occur during testing by re-executing the test cases that led to failures. Failures experienced in the field may be not easily reproducible, because of lack of execution details. To overcome these problems, we can augment applications with frameworks that capture the runtime data necessary to repeat the executions in house [7].

BCT does not depend on the implementation framework, except for monitoring. In this paper, we report our experience with Java. We recently extended BCT for C and C++, by substituting the Java monitoring framework with similar frameworks for C and C++ [39].

3 Model Synthesizers

In this section, we present the techniques to automatically generate models for program executions. As mentioned in the former section, we use *Daikon* to generate I/O models [16], and we propose *kBehavior* for incrementally synthesizing finite state automata that model component interactions.

Daikon accepts as input a set of samples composed of variables and associated values, and automatically generates predicates that are defined on the variables and are satisfied by all the samples. For instance, given a set of samples with variable `item.quantity` associated with positive values, Daikon generates the predicate `item.quantity>0`. To avoid discovering incidental predicates, that is predicates that hold because of limited available samples, Daikon computes statistical indexes that indicates the probability that predicates are incidental, and filters predicates whose indexes are above a given threshold. For instance, Daikon discovers the predicate `item.quantity>0` only when the number of samples is reasonably large.

kBehavior incrementally generates finite state automata from positive samples. We defined kBehavior, because the many available algorithms to generate finite state automata from sets of samples do not fit the requirements of our application, availability of only positive samples and incremental processing of samples to guarantee acceptable performances also for large complex component-based systems.

We monitor successful executions to derive a model of the component interactions, thus, the inference engine can only rely on positive samples, that is samples that must be included in the inferred model, and we cannot assume the availability of negative samples, that is samples of interactions that cannot be part of the inferred models. In general, we cannot assume the availability of any other source of information beside positive samples. For instance, we cannot assume the availability of knowledgeable teachers that can establish if an inferred FSA generates exactly the possible interactions, and samples are not available in a predetermined order. Thus, the many algorithms that rely on negative samples or any additional information badly adapt to our environment [9, 33].

Monitoring industrial size applications generates huge trace files that include long invocation sequences, and loading these traces into memory may require a huge amount of memory [3, 8, 35]. Algorithms that work on positive sample require access to the whole set of traces and thus are of limited efficiency [35,27].

Incremental algorithms that exploit repeated event patterns to obtain compact and general models can reduce memory consumption, because they do not require recording all traces. The reduction of memory consumption is particularly relevant when incrementally processing similar traces, that is traces that share many subsequences, as in the case of traces generated from component interactions. Thus, in our case, incremental algorithms should be particularly efficient. The only incremental algorithm that works with only positive samples is the *IID* algorithm [31]. The memory cost of this algorithm is $O(|\Sigma||P_l|N)$, where $|\Sigma|$ is the number of symbols in the alphabet, N the number of the states in the final automaton and $|P_l|$ is the size of a live-complete set for the final automaton [31]. In case of complex component-based systems, both the set of methods that can be invoked, that is $|\Sigma|$, and the live-complete sets, that is $|P_l|$, can be extremely large, and can lead to unacceptable inference time and memory consumption.

kBehavior works on positive samples only, without requiring additional information. It processes traces incrementally, and eliminates traces once processed, without requiring the availability of large memory space. It synthesizes automata efficiently by exploiting the similarities among traces, which are very common for traces that represent component interactions. It is more efficient and scales better than IID. Efficiency and scalability are obtained at the cost of convergence between the model generated from the samples and the correct model of the behavior. While IID guarantees that, when processing live-complete sets of samples, that is samples that cover all possible behaviors, the generated model tends to the correct one, kBehavior does not. Lack of convergence in presence of live-complete sets of samples is a theoretical, but rarely a practical problem, because experiencing a life-complete set of samples is a rare situation when monitoring the interactions of complex component-based system.

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

3.1 kBehavior

Let us consider the following three invocation sequences obtained by recording interactions with a Bank Account manager component:

Sequence 1:
```
createAccount() setCustomer() setAddress()
setTelephone() setInitialAmount()
activateAccount()
```

Sequence 2:
```
createAccount() setCustomer() setAddress()
setMobilePhone() setInitialAmount()
activateAccount()
```

Sequence 3:
```
createAccount() setCustomer() setAddress()
setMobilePhone() addAdditionalCustomer()
addAdditionalCustomer() addAdditionalCustomer()
addAdditionalCustomer() setInitialAmount()
activateAccount()
```

kBehavior processes sequences incrementally. It starts with the first one (`Sequence 1`), and builds an initial FSA that generates the first invocation sequence. Although in the general case that we will discuss in the next subsection, kBehavior can recognize simple recurrent patterns in the initial sequence, and model them in the FSA, in many simple cases the initial FSA maps the method calls to a linear sequence of transitions, as in the case of the example (trace 1 in Figure 2.)

When processing a new invocation sequence, kBehavior incrementally extends the current FSA by identifying invocation subsequences generated by subautomata and suitably extending the FSA by adding branches that connect the identified subautomata. For instance while processing the second invocation sequence (`Sequence 2`), kBehavior can detect that the simple FSA obtained from the first sequence already generates the prefix (createAccount() setCustomer() setAddress()) and the suffix (setInitialAmount() activateAccount()), and that in `Sequence 2` the two subsequences are connected with the invocation to method `setMobilePhone()`, while in the current FSA the corresponding subautomata are connected with transition `setTelephone()`. To include the new invocation sequence in the language generated by the FSA, kBehavior connects the subautomata that generate the identified subsequences with transitions that correspond to the invocations between these subsequences. In the example, kBehavior adds a transition `setMobilePhone()` between the final state of the subautomaton that generates the prefix and the initial state of the subautomaton that generates the suffix of `Sequence 2` (trace 2 in Figure 2.) When the subsequences that correspond to subautomata are connected by a long transition sequence, kBehavior is recursively executed on this sequence to identify an automaton that can be added to the current FSA instead of the corresponding linear transition sequence. For instance, the head and the tail of `Sequence 3` are connected with a sequence of invocations to method `addAdditionalCustomer()` that kBehavior can reduce to a simple looping automaton that can be added between the corresponding subautomata (trace3 Figure 2.)

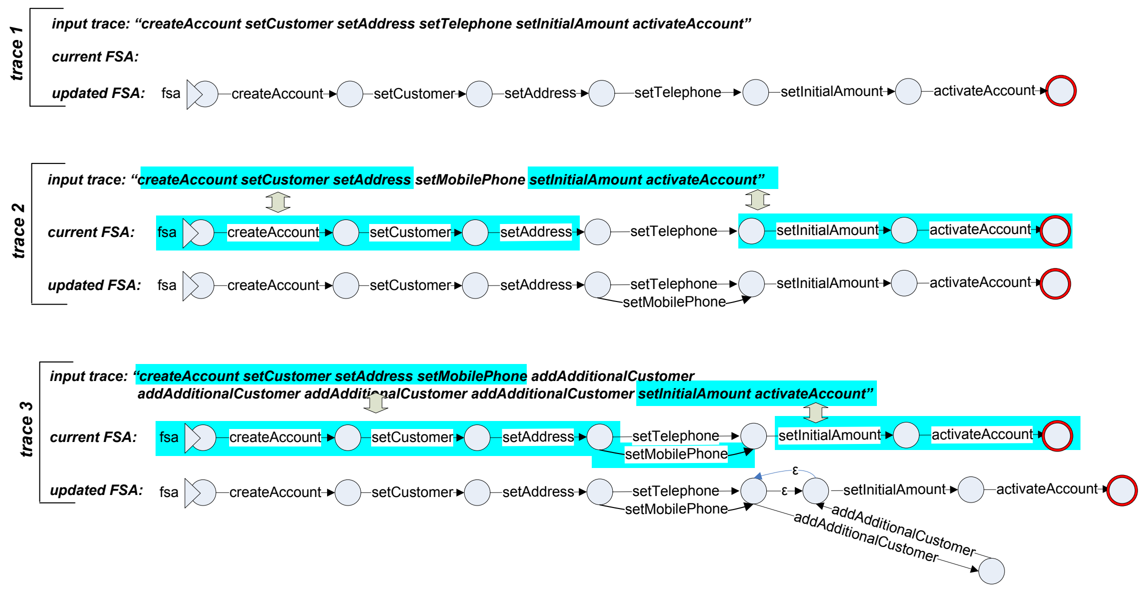

trace 1
input trace: "createAccount setCustomer setAddress setTelephone setInitialAmount activateAccount"
current FSA:
updated FSA:
fsa
createAccount
setCustomer
setAddress
setTelephone
setInitialAmount
activateAccount
trace 2
input trace: "createAccount setCustomer setAddress setMobilePhone setInitialAmount activateAccount"
setMobilePhone
trace 3
input trace: "createAccount setCustomer setAddress setMobilePhone addAdditionalCustomer addAdditionalCustomer addAdditionalCustomer addAdditionalCustomer setInitialAmount activateAccount"
ε
addAdditionalCustomer

Figure 2: Sample executions of the kBehavior algorithm.

3.2 Algorithm

To define kBehavior, we quickly recall some classic definitions of automata. A Non-Deterministic Finite State Automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite non-empty set of states,
- Σ is a finite non-empty set of input symbols or input alphabet,
- $\delta : Q \times \Sigma \rightarrow \wp(Q)$ is the transition function,
- $q_0 \in Q$ is the start state,

- $F \subseteq Q$ is the set of accepting states

If the function δ maps every pair $\langle$state, symbol$\rangle$ to a single state, that is $\delta : Q \times \Sigma \rightarrow Q$, 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:
$\delta^*(q, \lambda) = q$ where λ is the empty string,
$\delta^*(q, a\alpha) = \delta^*(\delta(q, a), \alpha)$ where $q \in Q$, $a \in \Sigma$, 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.

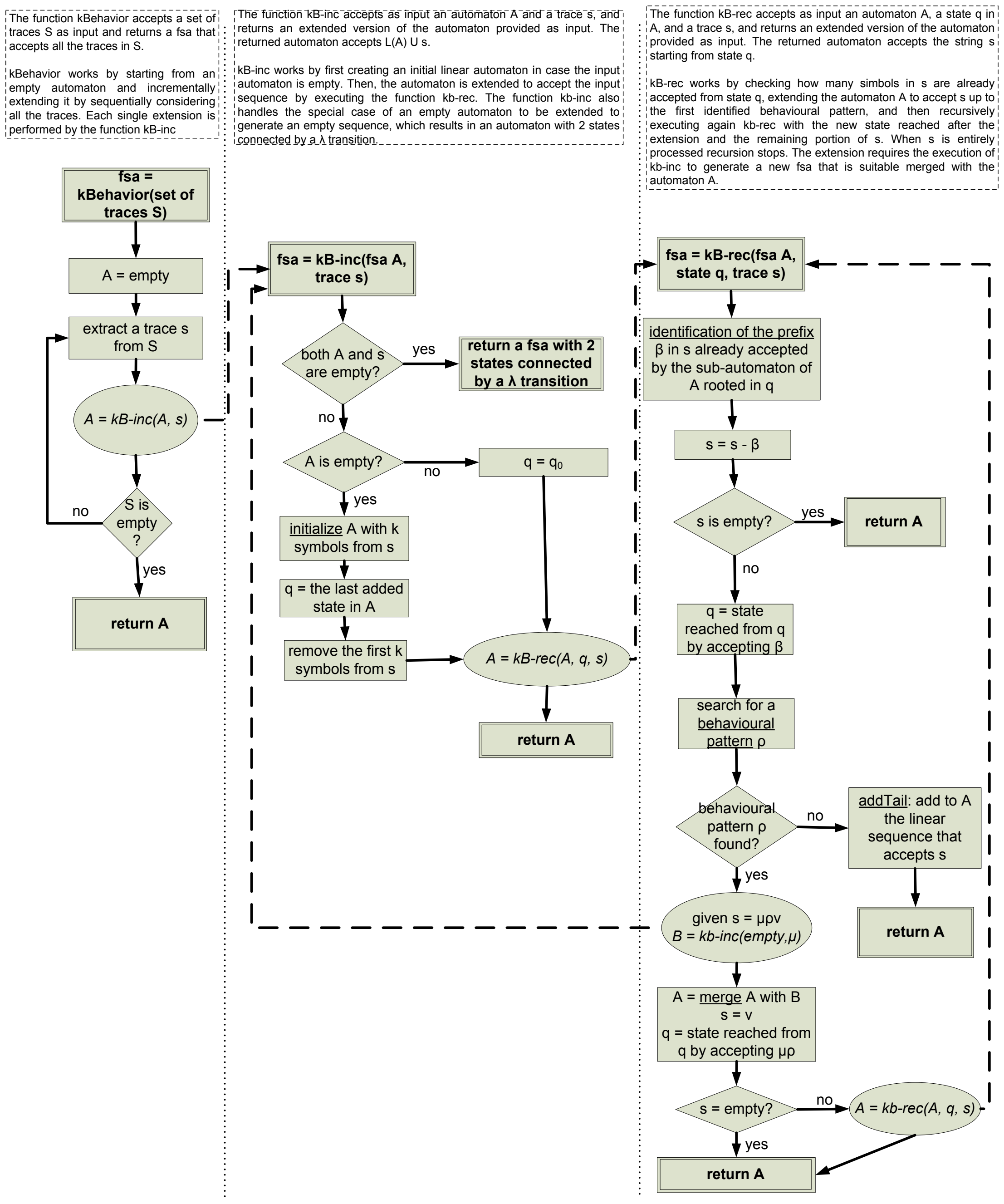
Given a set of strings S, *kBehavior* builds a finite state automaton A, whose language $L(A)$ includes S. kBehavior starts with an empty FSA, and processes strings incrementally. The incremental *kB-inc* step modifies the input automaton A to extend $L(A)$ with the input string s. When invoked with an empty automaton, kB-inc initializes the automaton to a simple linear automaton that generates the k-prefix of s, and removes the k-prefix from s. Then, kB-inc invokes the recursive step *kB-rec* that looks for sub-automata that generate substrings of s, and connects the identified sub-automata to include s in $L(A)$. Figure 3 illustrate the relation between the kBehavior algorithm and its components kB-inc and kB-rec, while Figures 4 and 5 summarize the algorithmic details.

Finding subautomata that generate substrings containing a single symbol of the alphabet is straightforward and not very interesting. To avoid identifying trivial subautomata, kBehavior limits the substrings that kB-rec tries to match with the FSA to a minimum length k, which is the main parameter of kBehavior.

kB-inc initializes the first empty automaton to the sequential automaton that generates the first k symbols of s. If s is shorter than k, kB-inc initializes the automaton to the sequential automaton that generates s. If s is empty, kB-inc initializes the automaton to the simple automaton with two states connected with a λ transition. If s is longer than k, kB-inc removes the first k symbols from s and invokes the recursive step kB-rec with the newly created linear automaton, the tail of the input string, and the initial state of the automaton.

kB-rec recursively processes an FSA A, a state q, and a string s in three main steps: (1) prefix identification, (2) behavioral pattern identification, and (3) extension.

In the *prefix identification* step, kB-rec identifies the longest prefix of s generated by A starting from q, removes the prefix from s, and changes the state q to the final state of the sub-automata that generates the prefix of s. In this way, Kb-rec records that the prefix is included in A, and thus it needs to extend A beyond the prefix, to generate the tail of s. For example, when kB-rec processes the second trace in the example in Figure 2, it identifies the prefix `createAccount() setCustomer() setAddress()` generated by the current FSA, and procedes with the tail `setMobilePhone() setInitialAmount() activateAccount()`. If the tail of s is empty, s is already generated by A, and


The function kBehavior accepts a set of traces S as input and returns a fsa that accepts all the traces in S.

kBehavior works by starting from an empty automaton and incrementally extending it by sequentially considering all the traces. Each single extension is performed by the function kB-inc

The function kB-inc accepts as input an automaton A and a trace s, and returns an extended version of the automaton provided as input. The returned automaton accepts L(A) U s.

kB-inc works by first creating an initial linear automaton in case the input automaton is empty. Then, the automaton is extended to accept the input sequence by executing the function kb-rec. The function kb-inc also handles the special case of an empty automaton to be extended to generate an empty sequence, which results in an automaton with 2 states connected by a λ transition.

The function kB-rec accepts as input an automaton A, a state q in A, and a trace s, and returns an extended version of the automaton provided as input. The returned automaton accepts the string s starting from state q.

kB-rec works by checking how many simbols in s are already accepted from state q, extending the automaton A to accept s up to the first identified behavioural pattern, and then recursively executing again kb-rec with the new state reached after the extension and the remaining portion of s. When s is entirely processed recursion stops. The extension requires the execution of kb-inc to generate a new fsa that is suitable merged with the automaton A.

fsa = kBehavior(set of traces S)
A = empty
extract a trace s from S
A = kB-inc(A, s)
S is empty ?
no
yes
return A
fsa = kB-inc(fsa A, trace s)
both A and s are empty?
yes
return a fsa with 2 states connected by a λ transition
no
A is empty?
no
q = q0
yes
initialize A with k symbols from s
q = the last added state in A
remove the first k symbols from s
A = kB-rec(A, q, s)
return A
fsa = kB-rec(fsa A, state q, trace s)
identification of the prefix β in s already accepted by the sub-automaton of A rooted in q
s = s - β
s is empty?
yes
return A
no
q = state reached from q by accepting β
search for a behavioural pattern ρ
behavioural pattern ρ found?
no
addTail: add to A the linear sequence that accepts s
return A
yes
given s = μρν
B = kb-inc(empty,μ)
A = merge A with B
s = ν
q = state reached from q by accepting μρ
s = empty?
no
A = kb-rec(A, q, s)
yes
return A

Figure 3: The logical steps of the kBehavior algorithm.

kB-rec returns to kBehavior that procedes with a new string. Otherwise, it procedes with the behavioral identification pattern step.

In the *behavioral pattern identification* step, kB-rec searches for a substring of s longer than k and generated by a sub-automaton of A. kBehavior works

```
Algorithm: kBehavior
Input: A FSA $A = (Q, \Sigma, \delta, q_0, F)$, a state $q$, and a string $\alpha$
         ($A$ is the current FSA, $q$ is the current state, and $\alpha$ is the string to be added to the language
         of $A$)
Output: the extended FSA that generates at least both all strings generated from the input FSA
          and the input string $\alpha$
begin
   (1) Initialization
     if $A$ == empty then initialize $A$ with the FSA that generates the first $k-1$ symbols of $\alpha$
     denoted with $\alpha_{k-1}$; then return kBehavior($A$,$\delta^*(q_0, \alpha_{k-1}), \alpha \setminus \alpha_{k-1}$)

     if $\alpha == \lambda$ then return the FSA with two states and one $\lambda$ transition connecting them; it is used
                               in the recursion

   (2) Identification of the prefix
     decomposition: $\alpha = \beta\gamma$ where $\beta$ is the longest prefix generated by $A$ from $q$
     if $\gamma == \lambda$ then return A, because $A$ already generates $\alpha$

   (3) Identification of the Behavioral Pattern $\rho$
     decomposition: $\gamma = \mu\rho\nu$, $A' \subseteq A$ s.t. $\delta^*(q'_0, \rho) \in F'$. Both the initial state and the set of
                    accepting states of $A'$ are arbitrarily defined from the set of states of $A$.
                    Several search strategies are possible for the decomposition. The algorithm
                    implements two possible strategies: searching for a decomposition that
                    either maximizes $|\rho|$ or minimizes $|\mu|$ with $|\rho| > k_\rho$.

     if $|\rho| < k$ then return addTail($A$, $\delta(q_0, \beta)$, $\gamma$)

   (4) Extension
     if $\delta(q_0, \beta) == q'_0$ then  /*incidental loop removal */
                                 $Q = Q \cup q'$
                                 $\delta$ is modified so that all incoming edges of $q$ become incoming
                                 edges of $q'$ and $q'$ is connected to $q$ by a $\lambda$-transition
                                 tmpFSA = kBehavior(empty, empty, $\mu$)
                                 A = merge(A, q', $\delta(q_0, \beta)$, tmpFSA)
     else
          tmpFSA = kBehavior(empty, empty, $\mu$)
          A = merge(A, $\delta(q_0, \beta)$, $q'_0$ tmpFSA)
   (5) Recursion
     if $\nu! = \lambda$ then go to (2) with $q_0 = \delta^*(q'_0, \rho)$ and $\alpha = \nu$
     else $F = F \cup \delta^*(q'_0, \rho)$ and exit
end
```

Figure 4: The kBehavior algorithm.

in two modes. In the *best matching* mode, kBehavior scans s sequentially and looks for the longest substring generated by a sub-automaton of A. This mode can be very expensive. In fact in the worst case, when the longest substring is the tail of s, kBehavior scans the whole string s, and searches for all possible sub-automata that generate substrings of s.

In the *optimized* mode, we introduce a second parameter $k_\rho \geq k$. kBehavior stops searching for substrings recognized by a subautomaton of s, when it finds a substring not shorter than k_ρ. As illustrated in Figure 6, the optimized mode not only reduces the impact of the worst case, but produces better results than the best matching mode. The best matching mode with $k = 2$ is shown in Figure 6(a): kB-rec correctly identifies the longest substring generated by a sub-automaton of the current FSA as the tail of the input string, and connects the sub-automaton that generates the head of the string with the identified sub-automaton, as informally discussed in the former section. In this way, kBehavior misses many similarities between the two paths connecting the two subautomata. The optimize mode with $k_\rho = k = 2$ is illustrated in Figure 6(b): kBehavior incrementally identifies three subautomata that generate substrings of the input string and produces a more compact FSA. Our experience indicates

```
Algorithm: merge
Input: the current FSA $A = (Q, \Sigma, \delta, q_0, F)$, the first state $q$, the end state $q'$ and the FSA
       to be merged $B = (Q', \Sigma', \delta', q'_0, F')$
Output: the FSA obtained by connecting $B$ with states $q$ and $q'$ of $A$
begin
    $Q = Q \cup Q'$
    $\delta$ is extended with transitions of $\delta'$
    $q'_0$ is merged with $q$
    $F'$ is reduced to one final accepting state that is merged with $q'$
    return $A$
end
```

```
Algorithm: addTail
Input: the current FSA $A = (Q, \Sigma, \delta, q_0, F)$, the state where the tail must be created $q$, and
       the behavior that the tail must generate $\pi$
Output: the FSA obtained by adding the tail to the state $q$
begin
    tmpFSA = a FSA composed of all transitions and states of $A$ that can be reached by $k - 1$
    steps from $q$.

    tmpFSA is incrementally extended with symbols in $\pi$. Each time a symbol is added, tmpFSA
    is checked for behavioral patterns. In that way, cycles are immediately detected and long
    tails are not created.

    return merging of tmpFSA with $A$
end
```

Figure 5: Structure of functions used in Figure 4.

that both the choice of $k_\rho = k = 2$ and of $k = 2$, and $k_\rho = 4$ are often effective.

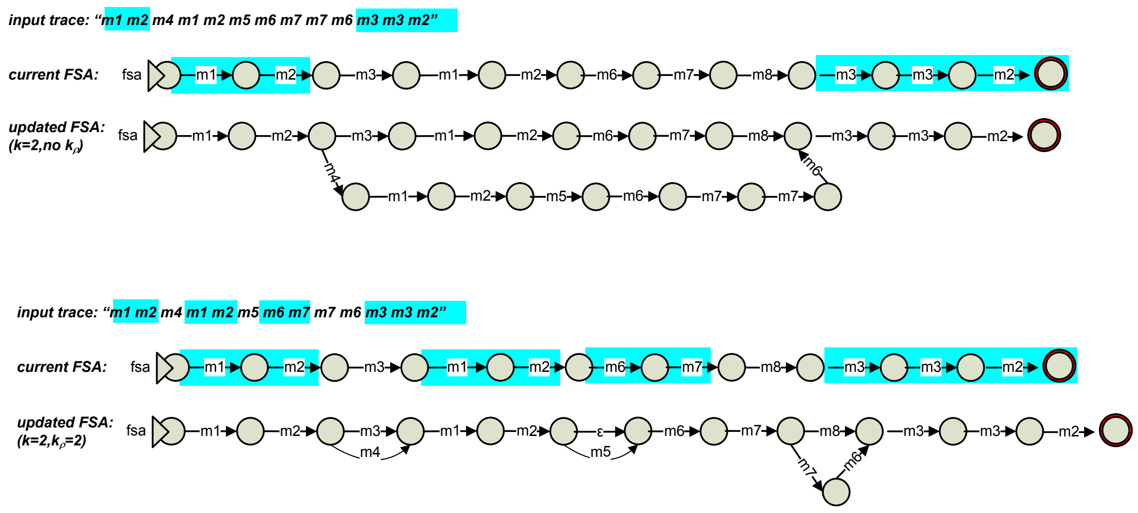

input trace: "m1 m2 m4 m1 m2 m5 m6 m7 m7 m6 m3 m3 m2"
current FSA:
fsa
updated FSA: (k=2,no k_ρ)
m1
m2
m3
m4
m5
m6
m7
m8
input trace: "m1 m2 m4 m1 m2 m5 m6 m7 m7 m6 m3 m3 m2"
current FSA:
updated FSA: (k=2,k_ρ=2)
ε

Figure 6: Effects of best matching and optimized behavioral pattern identification in kBehavior

After the behavior patter identification step, we have a state q that indicates the final state of the sub-automaton that generates the prefix of the original string s and a sub-automaton B that generates a substring b of the tail of the original string s. If b is not empty, in the *extension* step, kB-rec connects B to q, to extend the prefix of the original s generated by A. If b is empty, in the *extension* step, kB-rec connects q to an automaton that generates the current tail of s.

We first consider the case of b not empty. Let c be the part of the current s that precedes b. To correctly include s in $L(A)$, kB-rec connects q to B with an automaton C that generates c.

More precisely, kB-rec calls recursively kB-inc with the substring c to generate the automaton C. It then merges q with the initial state of C, and the final state of C with the initial state of B. C can be as simple as a pair of states connected with a λ transition, if c is empty, or a simple linear automaton if c is shorter than k, but can also be a non-trivial automaton. For example, when kB-rec processes the second trace in the example in Figure 2, it adds a transition *setMobilePhone()* between the sub-automaton that generates the prefix `createAccount() setCustomer() setAddress()` and the one that generates `setInitialAmount() activateAccount()`.

When q is the initial state of B the insertion of C by merging states creates a spurious loop. To avoid creating this spurious loop, kBehavior splits q into two states q_1 and q_2 connected with an empty (λ) transition, and merges q_1 with the initial state of C and q_2 with the final state of C. For example, the λ transition in Figure 6(b) derives from the splitting of the state to avoid spurious loops.

Once extended the current FSA with C, we recursively invoke kB-rec with the FSA extended with C, the final state of B as current state and the tail of s obtained by removing up to the prefix generated by B.

If b is empty, that is the behavioral pattern identification step does not identifies a substring of s generated by a sub-automaton, kB-rec recursively call kB-inc to create the automaton that generates the current s and merges the initial state of the automaton to q.

4 Automated Analysis of Violations

BCT addresses failures experienced in two main scenarios: during regression testing and in the field. When a failure is detected in any of these two cases, the failing execution is replayed in house and the inferred models are used to identify the anomalous events that can be responsible for the failure. Anomalous events consist of pairs (e, m), where e is a runtime event (either an interaction or a datum) that violates an inferred model m. Since each anomalous event corresponds to a model violation, in the rest of the paper, we will interchangeably use the terms anomalous events and model violations. BCT reports, properly filtered and aggregated, sets of model violations to testers who can use this information to identify the likely locations and causes of failures.

We automatically filter and aggregate violations for two main reasons:

- *Reason for automatically filtering anomalies*: Several violations can indeed be false positives; thus they do not represent faulty behaviors, but only executions that have never been observed before; manually inspecting all these violations is extremely time-consuming [37, 52].
- *Reason for automatically aggregating anomalies*: Multiple anomalies are frequently related, e.g., a unique problem may result in a model violation that transitively triggers further violations; a plain list of violations requires testers to waste time in inspecting a same problem multiple times from multiple (partial) point of views instead of analyzing a single organized and aggregated set of violations, which allows the inspection of a problem from a comprehensive point of view.

In the following, we present the two techniques to filter false positives and aggregate related anomalies. The final result is a graph that can be suitably and quickly investigated by testers.

Filtering False Positives

Since of the many ways of using systems, models can be easily violated without resulting in failures. The interesting violations are the ones directly related to failures. To automatically distinguish relevant violations from irrelevant violations with a high degree of confidence, we split violations in two sets according to the following heuristic: "violations observed during passed executions are likely unrelated with failures, even if also detected in a failing execution, because the system correctly served some requested functionalities independently from the existence of the anomalous events".

Given this criterion, it derives that violations observed during both failing and correct executions likely represent new uses of the system, not related to any experienced failure. Thus, they can be ignored. The remaining model violations, which are uniquely observed during failing executions, are indeed analyzed to find the cause of experienced failures. The empirical experiences reported in next Section show that this criterion has been particularly effective in filtering false positives when applied to our case studies. Similar techniques that consider the frequency of violations in faulty and correct executions to filter false positives have been successfully experienced in other work [26, 25].

Discovering Relations Between Violations

Usually violations do not occur in isolation, but a single problem is related to several model violations. For instance, an anomalous value can violate an I/O model and then generate an exception that, as side effect, causes violations of multiple interaction and I/O models, until the application recovers from the exception.

We discover relations by heuristically identifying sequences of violations related to a same problem and creating a graph-based representation of these relations, called *anomaly graph.* An anomaly graph can have one or more connected components, which are also graphs, that represent one or more issues experienced at different times of a same execution. In particular, each connected component consists of an acyclic graph that includes a coherent set of anomalous events related to a same problem. Anomalous events are represented as nodes, and likely cause-effect relations are represented with edges. The root of each connected component indicates the root violation that likely originated the problem represented by the component, and the different branches represent further violations likely caused by the previously observed ones. Note that acyclic connected components always have at least one root node. A simple anomaly graph that includes a single connected component is shown in Figure 8.

The technique for detecting likely related violations is based on the empirical observation that "violations related to a same problem are usually detected in methods executed from a (close) common ancestor method, i.e., the ancestor method executes a sequence of operations, one of them fails, and then the execution of the rest of the method generates further related violations". This may happen when an execution is compromised, and the exceptional flows cause several anomalous events until the execution either reaches a stable point or the application crashes.

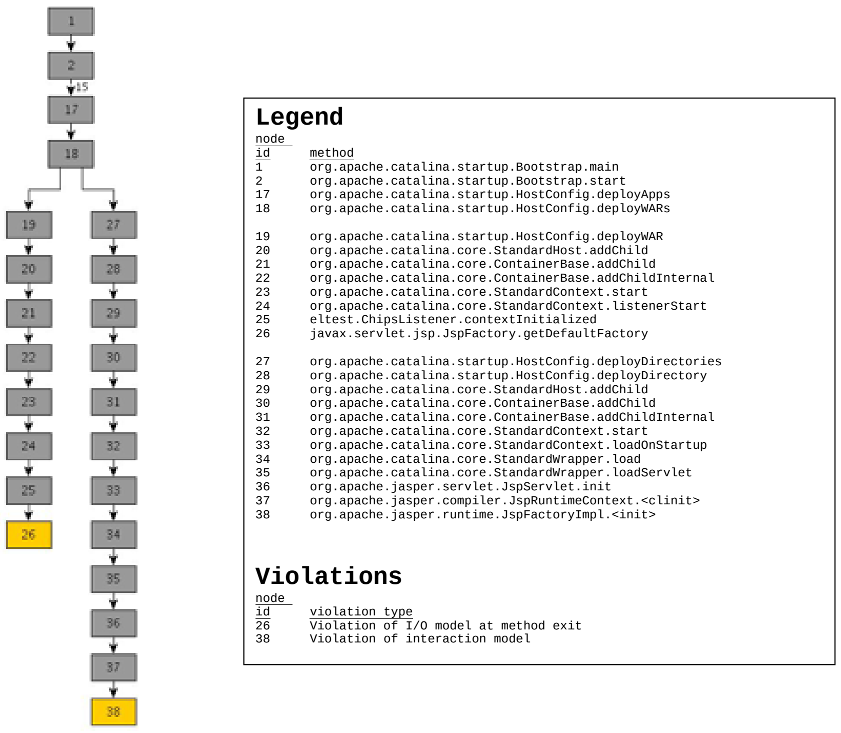
Anomaly graphs not only well address failures related to a single cause, represented with a single connected component, but also those failures requiring multiple (distant) anomalous events to be observed, e.g., state based faults or failures due to combination of multiple rare events. In fact, these failures would be captured with multiple connected components, one for each set of anomalous events.

Anomaly graphs are built in two steps. In the first step, the distance between violations is retrieved and represented in an initial connected acyclic graph. In the second step, the initial graph is refined into the anomaly graph.

Construction of the Initial Graph

An initial graph is created for each failing test case. Nodes represent violations[2] experienced in the execution of a failing test case. Edges indicate the ordering of violations, i.e., an edge connects violation A to B if A took place before B. Each edge is annotated with a value representing the distance between the violations connected by the edge. According to the heuristic discussed above, we measure the distance between two violations A and B on the dynamic call tree detected from the execution of the failing test case. A dynamic call tree is a tree that represents an observed execution where nodes indicate method

[2]in the construction of the initial graph, we only consider the violations not eliminated by the filtering false positives step.



Legend

node id	method
1	org.apache.catalina.startup.Bootstrap.main
2	org.apache.catalina.startup.Bootstrap.start
17	org.apache.catalina.startup.HostConfig.deployApps
18	org.apache.catalina.startup.HostConfig.deployWARs
19	org.apache.catalina.startup.HostConfig.deployWAR
20	org.apache.catalina.core.StandardHost.addChild
21	org.apache.catalina.core.ContainerBase.addChild
22	org.apache.catalina.core.ContainerBase.addChildInternal
23	org.apache.catalina.core.StandardContext.start
24	org.apache.catalina.core.StandardContext.listenerStart
25	eltest.ChipsListener.contextInitialized
26	javax.servlet.jsp.JspFactory.getDefaultFactory
27	org.apache.catalina.startup.HostConfig.deployDirectories
28	org.apache.catalina.startup.HostConfig.deployDirectory
29	org.apache.catalina.core.StandardHost.addChild
30	org.apache.catalina.core.ContainerBase.addChild
31	org.apache.catalina.core.ContainerBase.addChildInternal
32	org.apache.catalina.core.StandardContext.start
33	org.apache.catalina.core.StandardContext.loadOnStartup
34	org.apache.catalina.core.StandardWrapper.load
35	org.apache.catalina.core.StandardWrapper.loadServlet
36	org.apache.jasper.servlet.JspServlet.init
37	org.apache.jasper.compiler.JspRuntimeContext.<clinit>
38	org.apache.jasper.runtime.JspFactoryImpl.<init>

Violations

node id	violation type
26	Violation of I/O model at method exit
38	Violation of interaction model

Figure 7: The dynamic call tree corresponding to a failure in the Tomcat 6.0.4 application server (nodes from 3 to 16 are not displayed for brevity.

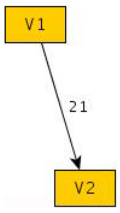
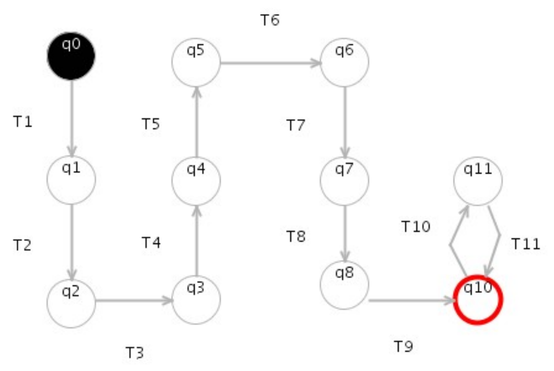


Figure 8: The anomaly graph corresponding to the problem experienced on Tomcat 6.0.4.

calls, and direct edges represent invocations, i.e., node A is connected to node B if method A directly invoked method B in the observed execution [2]. The distance between two violations associated with nodes A and B of a dynamic call tree is measured by computing the minimum number of edges that need to be traversed to move from A to a common ancestor method, and from the common ancestor method to B. For instance, Figure 7 shows the dynamic call tree corresponding to a failure experienced with Tomcat 6.0.4. Node 26 and 38 are associated with model violations. Figure 8 shows the initial graph derived from the Tomcat dynamic call tree.

The problem represented by the anomaly graph shown in Figure 8 can be observed during start-up of Tomcat 6.0.4 every time Tomcat is running web applications that use JSP factories during initialization. The fault in the Tomcat application server is a missing initialization that causes the erroneous propagation of a `null` value within the system (further information can be found at `http://issues.apache.org/bugzilla/show_bug.cgi?id=40820`). The initial graph automatically derived by BCT well represents the experienced problem. The root violation (node $V1$) is a violation of the I/O model `exit_returnValue != null` associated with method `javax.servlet.jsp.JspFactory.getDefaultFactory()`. This clearly indicates a failure in obtaining a factory object, which exactly represents and localizes the initialization problem in Tomcat. The violation $V2$ indicates the violation of an interaction model. This anomaly further clarify the cause of the failure. The anomaly indicates that the method `javax.servlet.jsp.JspFactory.<init>(()V` is unexpectedly invoked by method `org.apache.jasper.servlet.JspServlet.init((Ljavax.servlet.ServletConfig;)V)` when the execution reached state $q5$ in the automaton shown in Figure 9. The model indicates that the invocation to the static constructor of class `JspFactory` has been skipped. This missing invocation is exactly the cause of the failure.



Legend:
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;)

Figure 9: The FSA that generates anomaly V2.

The initial graph is immediately useful to testers only when the number of issues and corresponding violations is limited, such as for the Tomcat case study. In presence of large number of issues and violations, testers need to obtain a more focused view of the different problems that have been experienced and their relations. For instance, Figure 11 shows the initial graph for the Eclipse 3.3 case study, which is almost impossible to be manually interpreted by testers. Anomaly graphs are built from initial graphs to properly refine and isolate the different issues.

Generation of the Anomaly Graph

Our strategy for obtaining anomaly graphs from initial graphs is based on the removal of the edges that connect distant violations, thus representing relations that should be likely ignored. In case of small initial graphs, which are graphs with less than K nodes, we remove edges with weight less than $MaxWeight$. The resulting graph is the anomaly graph, and it may include one or more connected components corresponding to one or more problems that should be investigated by testers. According to our experiments (see next Section), effective values for these parameters are $K = 8$ and $MaxWeight = 15$.

In case of initial graphs with more than K nodes, the selection of a proper value for $MaxWeight$ can become critical because improper partitioning can generate either few, but large and incoherent, connected components or many, but tiny and fragmented, connected components. In both cases, the final result may be extremely difficult to be interpreted by testers.

We manage graphs with more than K nodes by selecting a proper value for $MaxWeight$ with a strategy inspired from clustering algorithms [18]. In particular, we consider the sequence of graphs obtained from the initial graph by using different values of $MaxWeight$ to remove edges, starting from a value corresponding to the highest weight associated with an edge down to 0. Then we evaluate the quality of the description provided by each graph by measuring the cohesion of its connected components. Since we incrementally remove the edges with the highest weight, cohesion of connected components is strictly increasing while $MaxWeight$ is decreasing. The best solution is identified by continuing to increase cohesion as long as moving from a graph to the following one introduces a significant improvement to cohesion. In other words, this search strategy intuitively corresponds to incrementally increasing the number of connected components (clusters) while benefits to cohesion are relevant. This rational is commonly used in the clustering algorithms based on Within Clustering Dispersion [18]. In the following, we provide details about our search strategy.

We start from a sequence of graphs $g_1, \ldots g_n$ obtained for the different values of $MaxWeight$, from the highest weight assigned to an edge to 0. We consider a graph as part of the sequence only if the number of connected components increases with respect to the previous graph. For instance, if a value M for $MaxWeight$ generates a graph g and a value $M-1$ generates a graph g' with the same number of connected components than g, g' is not part of the succession. These graphs are ignored because they do not contribute to clustering of violations and would only introduce noise in the input data of our algorithm.

For each connected component, we compute its inverse cohesion as the average value of the weights associated with its edges. Then we compute the inverse cohesion of the system as the average of the inverse cohesion of all its components.

$inv_c(C_j) = AVG_{edge_i \in edge(C_i)}(w_i)$, where C_j is a connected component, $edge(C_i)$ is the set of all edges in the graph C_i and w_i is the weight associated with edge i.

$inv_c(g) = AVG_{C_j \in comp(g)}(inv_c(C_j))$, where $comp(g)$ is the set of components in g.

The cohesion is defined as the inverse of the inverse cohesion:

$$cohesion(g) = \frac{1}{inv_c(g)}.$$

It is clear that removing all edges produces the highest cohesion for each connected component, which would be formed by single nodes only. Thus, the cohesion of the connected components, and their average as well, is strictly increasing (and the inverse cohesion is strictly decreasing), while decreasing the value of $MaxWeight$.

The difference between the inverse cohesion of two consecutive graphs in the sequence indicates the gain in moving from one graph to the following. In particular, given a graph g_1 obtained with $MaxWeight = w_1$ and g_2 obtained with $MaxWeight = w_2 < w_1$, the gain from g_1 to g_2 is given by $gain(g_1, g_2) = \frac{inv_c(g_1) - inv_c(g_2)}{w_1 - w_2}$. To shorten the notation, we denote $gain(g_1, g_2)$ as $gain_{1,2}$. Given a sequence of gain values $gain_{1,2}, gain_{2,3}, gain_{3,4}, \ldots gain_{m-1,m}$, obtained from a sequence of graphs $g_1, g_2, \ldots, g_m$, we indicate the current trend in the cohesion of the connected components as $trend_i = |gain_{i,i+1} - gain_{i-1,i}|$, with $i \in 2, \ldots m-1$. We select the best solution to be presented to testers by identifying the graph G_I associated with the highest value of the trend, i.e., if $trend_I \geq trend_i \forall i = 1..m$.

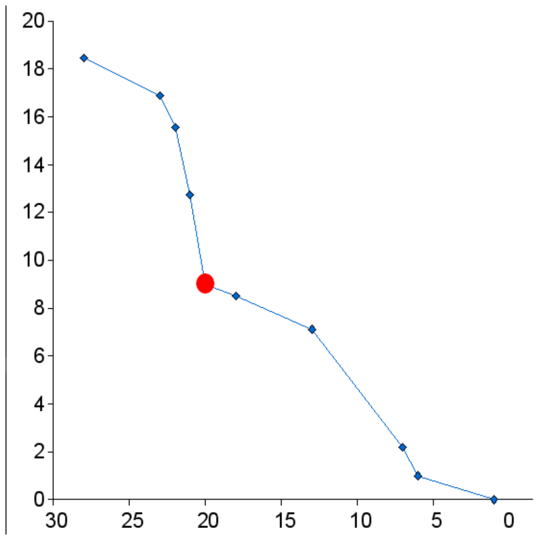


Figure 10: The trend for the Eclipse 3.3. case study.

We can give a visual interpretation to our strategy. If we plot in a graph the value of the average inverse cohesion and the value of $MaxWeight$, the gain is given by the slope of the lines connecting the different points in the graph. The best configuration C_I can be found in correspondence to the point in the graph that induces the biggest difference in the trend of the gain, i.e., a big gain in the previous step, followed by a small gain in the successive step. For instance, Figure 10 shows this graph for the Eclipse 3.3 case study presented in next session. The point with the circle indicates the best configuration identified by our algorithm. In fact, the selected configuration actually corresponds to the best partitioning of the detected violations because it consists of 11 connected components that properly represent the different sets of related violations. Figure 11 shows the initial graph and Figure 12 shows the final anomaly graph.

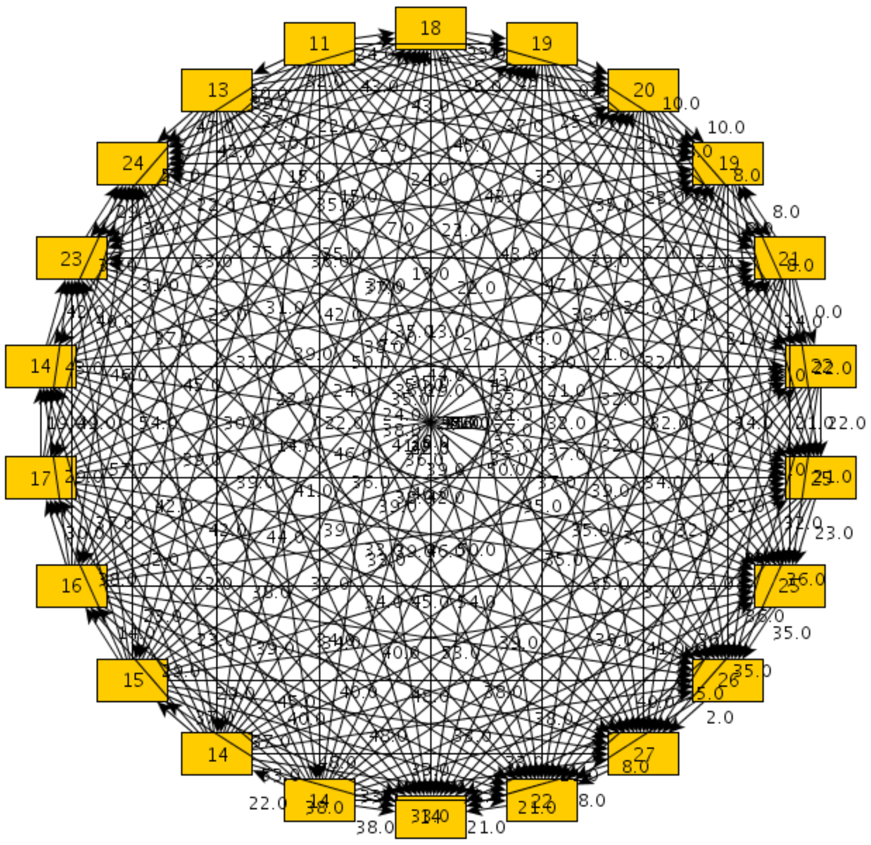


Figure 11: The initial graph corresponding to the problem experienced with Eclipse 3.3.

Since false positives often occur in isolation, we present to testers the different connected components ordered with respect to their size, from the biggest to the smallest. For instance, in the case of the Eclipse 3.3 case study we have 11 connected components, but the two relevant connected components describing the experienced problem are presented at the top of the sequence. The remaining connected components, that include several false positives, can be ignored by testers because the first two are sufficient to explain the experienced failure.

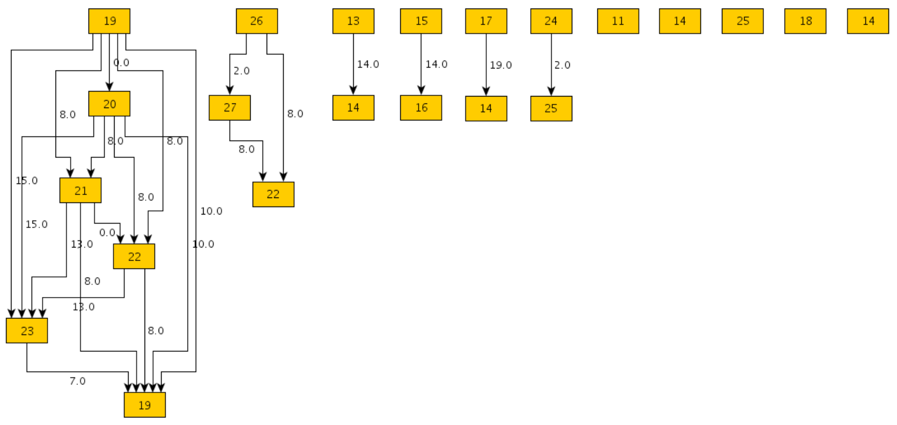


Figure 12: The anomaly graph corresponding to the problem experienced with Eclipse 3.3.

5 Validation

Data are extracted from running systems by using monitoring platforms. We integrated the BCT technology with two solutions: Aspectwerkz [1], which is an aspect-oriented framework [23], and the TPTP probe technology, which is part of the TPTP platform [13]. Aspectwerkz and TPTP probes are similar: they allow to extract runtime data from Java systems without requiring availability of source code and the set of captured events consist of method invocations. BCT captures all inter-component method invocations, while it ignores internal computations.

The objective of the validation presented in this Section is twofold: (1) to show that automated analysis of violations provides results that point to failure causes and locations, and are easier to be analyzed than results provided by the existing anomaly detection techniques that do not post-process the detected anomalies, and (2) to show that results provided by BCT can be effectively used by testers who spend no time, or a minimal amount of time in the worst case, in inspecting false alarms.

Techniques for anomaly detection generate models that are close to the models generated by BCT. For example, Wasylkowski et al. use inferred FSA to detect anomalies in interactions between objects [48] and Raz et al. use inferred I/O models to detect anomalies in data exchanged with on-line services [34]. BCT advances these techniques by both filtering the many false positives that would be otherwise presented to testers and aggregating the remaining violations in a coherent and structured way, so that testers do not spend time in unnecessarily inspecting multiple violations related to a same problem.

The empirical work confirms that these results hold in regression testing and analysis of in-the-field problems. In our validation, we considered three systems of different size and complexity: NanoXML [38], Eclipse [12] and Tomcat [41]. NanoXML is a XML parser of about 6752 lines of code. Eclipse is a development platform extendible with plug-in provided by third parties (COTS components). The Eclipse configuration considered in our experiments consists of about 3960323 lines of code (we counted the lines of code of both the eclipse core components and the installed plugins). Finally, Tomcat is an application server of about 197343 lines of code. Even if several parts of these systems are provided with source code, we analyzed components without exploiting its availability and without any knowledge about the analyzed components, thus de facto working with COTS components. Source code has been only exploited to verify that the feedback provided by BCT corresponds to actual faults. In the following, we describe the empirical experiences in detection of fault causes and locations, for regression testing and analysis of in-the-field failures.

Regression Testing

Effectiveness of BCT in detecting location and cause of regression problems has been evaluated against fault-introducing updates on NanoXML, Tomcat and Eclipse. In the case of NanoXML, faults have been injected by third parties [17], while for Tomcat and Eclipse faults are real problems experienced with released versions of the software. In case of NanoXML and Eclipse, test cases are provided together with applications, while for the Tomcat application server we designed test cases that exercise the web application manager according to the category partition method [30].

We measure effectiveness of BCT by computing two indexes: the percentage of hidden false positives and the percentage of connected components in anomaly graphs that need to be inspected to identify the cause and location of an experienced problem. The former indicates the capability of BCT to let testers concentrate on useful information only, while the latter represents how much the information presented to testers is useful to identify faults.

Results are summarized in Table 1. The first column indicates the case study considered in the empirical experience. The names assigned to Nano XML case studies exactly match the names assigned to the different versions of the application in the SIR repository [17], we run the provided test cases and used BCT to identify the anomalies. The Tomcat case study regards a regression fault introduced during an update from version 5.5.12 to version 5.5.13: with Tomcat 5.5.13 the reload of a web application causes a NullPointerException to be thrown, the problem was not present in version 5.5.12 of the server [43]. The Eclipse case study regard fault introduced in the update of EMF from version v200606150000 to v200701181044 within Eclipse 3.3: the fault causes a JUnit test failure [11]

The remaining portion of the table is structured in two main parts. The first part provides data about effectiveness of BCT in presenting useful information, while the second part provides data about effectiveness of BCT in filtering useless data.

The column "Num CC" indicates the number of connected components in the anomaly graph. Testers inspect the connected components starting from the largest one to the smaller one. The column "Inspected CC" indicates the number of connected components that need to be inspected before identifying the components responsible for the failure and the cause of the experienced problem. The value within `()` indicates the number of connected components that do not carry useful information and are inspected by testers, thus causing loss of time.

The second part of the table provides information about anomalous events that have been automatically filtered. The column "Num Meaningful" indicates the relevant violations that have been erroneously discarded. The column "False Positives" indicates the number of model violations carrying no information that have been automatically eliminated from the analysis. Finally, the column "Perc of Filtered False Positives" indicates the percentage of irrelevant violations that have been automatically eliminated from the analysis.

We can state that BCT suitably supported testers in identifying the cause and location of the experienced problems in most of the case studies. In fact, connected components suitably described the failure, and the corresponding interactions causing the failure, in 8 out of 10 case studies. The two unsuccessful case studies have been Nano XML v4 to v5 F_XER_HD_1 and Nano XML v4 to v5 F_NV_HD_1, where a single false alarm is inspected by testers. On the other hand, the 8 successful case studies have been effectively completed. In fact, all problems have been identified inspecting at most 3 connected components and a limited number of connected components that do not provide relevant information have been presented to testers (1 in the worst case).

The two unsuccessful case studies provide an interesting observation. Even when BCT is not able to determine the cause and location of an experienced problem, testers do not have to spend time with ineffective information. In fact, in both case studies 95% of the false alarms have been automatically filtered from the analysis and only 1 connected component has been presented to testers, who can quickly examine and discard it.

The capability to filter false positives is not restricted to the unsuccessful case studies. In fact in most cases BCT filtered more than the 90% of the false positives and the average number of filtered false positives is 89%.

	CC Presented to Testers		Hidden Anomalies		
Case Study	Num. CC	Inspected CC (inspected FP)	Num Meaningful	False Positives	Perc of Filtered False Positives
Nano XML v5, all_f	3	3 (0)	0	4	100%
Nano XML v4 to v5, SR_HD_1	1	1 (0)	0	19	100%
Nano XML v4 to v5, F_XER_HD_1	1	1 (1)	0	18	95%
Nano XML v4 to v5, F_CR_HD_3	1	1 (0)	0	17	100%
Nano XML v4 to v5, F_NV_HD_1	1	1 (1)	0	21	95%
Nano XML v4 to v5, all_f	6	2 (0)	0	16	80%
from Tomcat 5.5.12 to 5.5.13	2	1 (0)	0	2	67%
Eclipse 3.3, GMF, EMFT, OCL	11	2 (0)	0	3	75%

Table 1: Summary of the Experiments About Regression Testing

Let us note that testers would have to pay an extremely high cost for inspecting large sets of model violations without using techniques to filter false positives and aggregate related anomalous events. In fact, false positives represent a large portion of the detected model violations. Moreover inspection of a high number of true positive violations can be complex and time consuming as well, if not using aggregation and prioritization techniques. Consider for instance the time necessary to inspect the violations detected for the Eclipse case study shown in Figure 11 instead of the automatically generated and prioritized set of connected components shown in Figure 12.

In Field Failures

Effectiveness of BCT in detecting location and causes of in field failures has been evaluated against case studies related to Eclipse and Tomcat. All the considered case studies address problems experienced with the released versions of the applications, thus failures experienced by several real users of these systems. Table 2 shows the results of the application of BCT to field failures.

The column "case study" indicates the target case study. The "ProbeKit, Eclipse(chmod)" indicates a known issue that affects Eclipse TPTP 4.4 on Linux: when installing TPTP through the Eclipse Update procedure TPTP Probekit instrumenter is not set as beeing executable, this make instrumentation not possible, and causes the generation of error messages [45]. The "ProbeKit, Eclipse(EM64T)" indicates a fault that affect Eclipse TPTP 4.4 when it is run on 64 bit architectures: TPTP libraries for 64 architectures are not provided with TPTP, this makes Probe instrumentation not possible and causes the generation of error messages shown in popup windows and logs [44]. The "EMF

2.2.1, WTP" case study regard a know incompatibility between EMF 2.2.1 and WTP 1.5.1 that caused several user problems [14,46]. Finally, the Tomcat 6.0.4 indicates a fault experienced with Tomcat version 6.0.4: a web application is not started at server boot because of a NullPointerException [42].

The column "Num CC" indicates the total number of connected components presented to testers. The column "Inspected CC" indicates the number of connected components inspected by testers to identify the location and cause of the experienced problem. The number between () indicates the number of connected components that have been inspected and that carry no information.

We can note that 3 of the 4 empirical experiences have been successfully completed with limited effort for testers, at most 3 connected components were inspected to identify the problem. Moreover, also for in field failures, the number of false positives is extremely limited: 0 for successful case studies and 2 for the unsuccessful one. Thus, confirming the effectiveness of BCT in limiting the number false alarms, even in the unsuccessful test cases.

	Issues Presented to Testers	
Case Study	Num. CC	Inspected CC (inspected FP)
ProbeKit, Eclipse(chmod)	1	1 (0)
ProbeKit, Eclipse(EM64T)	1	1 (0)
EMF 2.2.1, WTP	2	2 (2)
Tomcat 6.0.4	3	3 (0)

Table 2: Summary of the Experiments About Field Failures

6 Conclusion

Software systems that integrate grey-box components, i.e. components that come with partial or missing specifications and source code, are typically difficult to debug and analyze: missing information often make static analysis and debugging techniques not applicable. Dynamic analysis techniques demonstrated to be effective for fault localization even when grey-box components are used: these techniques monitor components interfaces and localize faults by identifying violations of automatically inferred models. Usually these techniques work in three phases: the training phase in which the system is monitored to collect execution data, the inference phase in which models are automatically inferred from collected data and finally the checking phase in which the system behavior is validated in the field by comparing runtime data with the inferred models.

Unfortunately also dynamic analysis techniques present some limitations. Most of them focus on particular aspects like data-values or interactions protocols and do not integrate such information thus limiting the effectiveness of their analysis. Many of the existing techniques suffer from scalability issues: they store huge amount of data thus making monitoring of complex system costly, moreover they do not help developers in case a huge amount of violations and false positives are identified.

In this paper we presented *Behavior Capture and Test (BCT)* a methodology that overcomes the limitations of the existing dynamic analysis techniques. BCT automatically derives behavioral models from execution data recorded at testing time. BCT produces models that capture complementary aspects of components interactions by integrating data invariants derived with Daikon with finite state models inferred with k-behaviour, an incremental inference engine that generalizes interaction sequences. Furthermore BCT permits to incrementally infer models thus reducing space consumption.

BCT localizes faults by monitoring system executions and by comparing the application behavior with the inferred models: model violations indicate behavioral anomalies and help developers in localizing faults and identifying their causes. In order to increase the effectiveness of results BCT prunes anomalies occurring both in valid and failing executions thus reducing the number of false positives. The result of BCT analysis is an anomaly graph in which nodes represent violations and oriented edges causal relations among them: BCT clusters related anomalies using a Within Clustering Dispersion approach and presents the final anomaly graph to testers. Connected components are then inspected by size: since faults often cause many anomalies, developers inspect big clusters of related anomalies first, in this way eventual false positives can be further eliminated.

We validated the technique by applying BCT to injected and real faults affecting NanoXml, Eclipse and Tomcat. We validated the technique considering two types of faults that typically affect component-based applications: regression faults caused by the update of one ore more components and faults identified in the field when final users stress complex interactions not covered at testing time. The obtained results highlight the effectiveness of BCT: the inferred models permitted to identify behavioral anomalies useful to identify fault causes, while the automated anomaly analysis reduced the effort required for the diagnosis. BCT succesfully identified fault locations and causes in 9 of the

12 case studies considered. The automated identification and removal of false positives filtered out from 67% to 100% of false positives in each case study. Finally the anomaly graphs presented to testers permitted them to identify faults by analyzing only a few of the remaining anomalies.

Acknowledgment

This work has been partially supported by the European Community under the Information Society Technologies (IST) programme of the 6th FP for RTD - project SHADOWS contract IST-035157.

References

[1] Aspectwerkz. `http://aspectwerkz.codehaus.org/`, visited in 2008.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*. ACM, 1997.

[3] A. Biermann and J. Feldman. On the synthesis of finite state machines from samples of their behavior. *IEEE Transactions on Computer*, 21:592–597, June 1972.

[4] L. Briand, Y. Labiche, and X. Liu. Using machine learning to support debugging with tarantula. In *proceedings of the International Symposium on Software Reliability Engineering*, 2007.

[5] CADology. CAD software tools. `http://www.cadology.com`.

[6] S. Chandra and P. M. Chen. Whither generic recovery from application faults? A fault study using open-source software. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks / Symposium on Fault-Tolerant Computing (DSN/FTCS)*, 2000.

[7] J. A. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *proceedings of the International Conference on Software Engineering*. IEEE Computer Society, 2007.

[8] J. Cook and A. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[9] P. Dupont. Incremental regular inference. In L. Miclet and C. Higuera, editors, *proceedings of the 3rd International Colloquium on Grammatical Inference*, volume 1147 of *LNCS*. Springer-Verlag, 1996.

[10] Eclipse Bugzilla. Bug reports for eclipse project. `https://bugs.eclipse.org/bugs/`. Visited in 2008.

[11] Eclipse Fault. Eclipse bugzilla report for bug id 181288. `https://bugs.eclipse.org/bugs/show_bug.cgi?id=181288`. Visited in 2008.

[12] Eclipse Foundation. Eclipse. `http://www.eclipse.org/`.

[13] Eclipse TPTP. Eclipse test & performance tools platform. `http://www.eclipse.org/tptp/`, visited in 2008.

[14] Eclipse WTP. Wtp 1.5.0 requirements. `http://archive.eclipse.org/webtools/downloads/drops/R1.5/R-1.5.0-200606281455/`. Visited in 2008.

[15] S. Eldh, S. Punnekkat, H. Hansson, and P. Jöonsson. Component testing is not enough - a study of software faults in telecom middleware. In *19th IFIP International Conference on Testing of Communicating Systems*, LNCS. Springer, 2007.

[16] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[17] Galileo Research Group. Software-artifact infrastructure repository (SIR). `http://esquared.unl.edu/sir`.

[18] A. Gordon. *Classification.* Chapman and Hall/CRC, 2 edition, 1999.

[19] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *proceedings of the 24th International Conference on Software Engineering.* ACM Press, 2002.

[20] S. Hissam and D. Carney. Isolating faults in complex COTS-based systems. SEI Monographs PA 15213-3890, Carnegie Mellon Software Engineering Institute, 1998.

[21] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *proceedings to the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2004.

[22] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *proceedings of the Internation Conference on Software Engineering*, 2002.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *proceedings of the European Conference on Object Oriented Programming*, 1997.

[24] N. G. Leveson. The role of software in spacecraft accidents. *AIAA Journal of Spacecraft and Rockets*, 41(4), 2004.

[25] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 2005.

[26] C. Liu and J. Han. Failure proximity: a fault localization-based approach. In *proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 2006.

[27] L. Mariani and M. Pezzè. Dynamic detection of cots components incompatibility. *IEEE Software*, 24(5):76–85, September/October 2007.

[28] W. Masri, A. Podgurski, and D. Leon. An empirical study of test case filtering techniques based on exercising information flows. *IEEE Transactions on Software Engineering*, 33(7):454–477, July 2007.

[29] 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. In *proceedings of the 22nd International Conference on Software Engineering.* ACM, 2000.

[30] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[31] R. Parekh, C. Nichitiu, and V. Honavar. A polynomial time incremental algorithm for learning dfa. In *proceedings of the International Colloquium on Grammatical Inference (ICGI)*, volume 1433 of *LNCS*. Springer, 1998.

[32] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *proceedings of the 12th ACM SIGSOFT twelfth International Symposium on Foundations of Software Engineering*, pages 23–32. ACM, 2004.

[33] S. Porat and J. Feldman. Learning automata from ordered examples. *Machine Learning*, 7:109–138, 1991.

[34] O. Raz, P. Koopman, and M. Shaw. Semantic anomaly detection in online data sources. In *proceedings of the 24th International Conference on Software Engineering*. ACM Press, 2002.

[35] S. P. Reiss and M. Renieris. Encoding program executions. In *proceedings of the 23rd International Conference on Software Engineering*, pages 221–230. IEEE Press, 2001.

[36] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *proceedings of the Internation Conference on Automated Software Engineering*, 2003.

[37] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE)*, 2004.

[38] M. D. Scheemaecker. Nano XML.

[39] O. Spinczyk, M. Urban, D. Lohmann, G. Blaschke, and R. Sand. Aspect c++. `http://www.aspectc.org`, visited in 2008.

[40] Sun. Java reflection. `http://java.sun.com/javase/6/docs/technotes/guides/reflection/`, visited in 2008.

[41] The Apache Software Foundation. Tomcat. `http://tomcat.apache.org/`.

[42] Tomcat Fault. Tomcat bugzilla report for bug id 40820.

[43] Tomcat Fault. Tomcat bugzilla report for bug id 41939. `http://issues.apache.org/bugzilla/show_bug.cgi?id=41939`. Visited in 2008.

[44] TPTP Fault. Eclipse tptp bugzilla report for bug id 157486. `https://bugs.eclipse.org/bugs/show_bug.cgi?id=157486`. Visited in 2008.

[45] TPTP Fault. User report for an eclipse tptp issue. `http://dev.eclipse.org/mhonarc/lists/tptp-tracing-profiling-tools-dev/msg00305.html`. Visited in 2008.

[46] Ubuntu mailing list. User report for an issue caused by the emf 2.2.1 - wtp 1.5.0 incompatibility. `http://ubuntuforums.org/archive/index.php/t-307526.html`. Visited in 2008.

[47] 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. In *15th International Symposium on Software Reliability Engineering (ISSRE'04).*, 2004.

[48] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *proceedings of the joint meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2007.

[49] E. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, September/October 1998.

[50] A. Zeller. Isolating cause-effect chains from computer programs. In *proceedings of the Symposium on the Foundations of Software Engineering*, 2002.

[51] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufman, 2005.

[52] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2004.