Automated Identification of Failure Causes in System Logs

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

Abstract

Log files are commonly inspected by system administrators and developers to detect suspicious behaviors and diagnose failure causes. Since size of log files grows fast, thus making manual analysis impractical, different automatic techniques have been proposed to analyze log files. Unfortunately, accuracy and effectiveness of these techniques are often limited by the unstructured nature of logged messages and the variety of data that can be logged.

This paper presents a technique to automatically analyze log files and retrieve important information to identify failure causes. The technique automatically identifies dependencies between events and values in logs corresponding to legal executions, generates models of legal behaviors and compares log files collected during failing executions with the generated models to detect anomalous event sequences that are presented to users. Experimental results show the effectiveness of the technique in supporting developers and testers to identify failure causes.

1 Introduction

In many software systems, runtime data are continuously collected in the field to trace executions. These data are typically stored in log files, and analyzed in case of system failures and malfunctions, to identify causes and locations of problems. Logs are particularly useful when systems are large, concurrent and asynchronous, because their execution space is hard to sample at testing time, and several untested, and sometimes illegal, event sequences may be observed only when final users interact with applications.

Since data are massively collected from executions, the size of log files can grow fast [24]. This is particularly true when logging collects not only the observed event sequences, but also parameters associated with events. For instance, many e-commerce applications trace IP numbers and URLs of accessed Web pages to create user profiles and recognize the action sequences responsible for failures [6].

When problems occur, debugging starts from the inspection of log files. Size, heterogeneity and complexity of logs seldom make manual inspection cost-effective. In addition, system administrators and developers usually do not have a complete knowledge of target systems, thus a large amount of time may be necessary to manually isolate suspicious event sequences.

Automatic techniques have been proposed to support log file analysis. Techniques can be grouped into three main classes: specification-based techniques, expert systems and heuristic based techniques. Specification based-techniques match events in log files with formal specifications that describe legal event sequences [1]. The recognized anomalies, i.e., event sequences not accepted by specifications, are presented to testers. These techniques have the important benefit to present only and all the problems that can be detected from logs. Unfortunately, complete and consistent specifications are expensive to be produced and maintained. Thus, suitable specifications are seldom available and it is rarely possible to apply such approaches.

Expert systems do not require complete specifications to identify suspicious sequences, but require user-defined catalogs that describe the events that are commonly related to failures [17]. Since the commonly used log formats (e.g. syslog [22], java simple log format [29], uniform log format [28]) include unstructured data, expert systems for log file analysis require to be tailored with user-defined regular expressions to analyze and recognize interesting event types [19, 31, 14]. The usually large number of possible event types makes the definition and maintenance of such regular expressions an expensive and error prone task [26]. Similarly to expert systems, symptom databases use known patterns of illegal event sequences to detect failure causes [7]. Unfortunately, maintaining symptom databases up-to-date is expensive and their effectiveness is lim-

1071-9458/08 $25.00 © 2008 IEEE
DOI 10.1109/ISSRE.2008.48

ited to well-known and documented problems, and cannot help in case of undocumented issues.

Heuristic based approaches is the class of techniques that provide the most general solution and require little effort to be applied [5, 6, 32, 27]. These techniques detect legal and illegal event sequences by using supervised and unsupervised machine learning algorithms [13]. Supervised learning techniques analyze log files corresponding to failing and successful executions to extract enough knowledge to automatically detect issues in future executions [6]. In the learning phase, user intervention is required to distinguish legal and illegal executions.

Unsupervised approaches fully automate the analysis process and automatically detect clusters of related events from log files [18, 32]. Suspicious events are identified by selecting the events that do not belong to any cluster (also known as outliers) [27].

High automation is the major benefit of heuristic-based approaches, but their effectiveness is limited by expressiveness of learned models. Existing techniques suitably capture faults due to single unexpected events, but do not address problems related to specific sequences, e.g., a set of events that are individually acceptable but lead to a problem if they occur in an unexpected order, or specific data flows, e.g., data attributes that are individually acceptable but lead to problems if they appear in an unexpected order.

In this paper, we propose a heuristic-based technique that automatically analyzes logs and identifies problems not only depending from single events, but also from specific event sequences and data flows. The technique works by deriving general and compact models of the expected behavior from logs recorded during legal executions, in the form of finite state automata; comparing logs collected during failing executions with these models; and presenting to testers the likely illegal events, together with a representation of the behavior which has been usually accepted by the system. This output is a valuable information for testers, who can quickly identify fault locations and causes.

The technique presented in this paper extends a previous work [8] and provides the following specific technical contributions:

- it defines strategies to handle concrete data values and automatically discover data-flow information from attributes associated with events;
- it derives models that support the discovery of precise illegal event sequences juxtaposed to expected legal sequences, on the contrary of existing automated log file analysis techniques that only indicate suspicious sets of events [5];
- it defines a multi-model technique that allows to analyze a same problem at different levels of granularity (complete application, single components or user actions);
- it combines automata learning and data clustering approaches to analyze different formats of log files;
- it is applied to a set of case studies that show the effectiveness of the approach.

The paper is organized as follow. Section 2 overviews the heuristic-based log file analysis technique presented in this paper. Section 3 presents a technique for automatically separating event names from attribute values. Section 4 describes how to handle data associated with events. Section 5 presents the technique for generating models from sets of legal executions. Section 6 presents the technique for comparing logs collected during failing executions with automatically generated models. Section 7 describes empirical results obtained with several case studies. Section 8 discusses related work. Finally, Section 9 summarizes the paper and discusses future work.

2 Log File Analysis

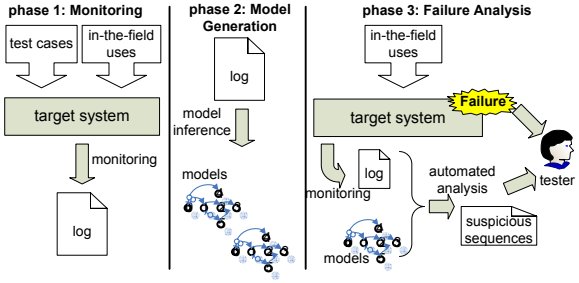


Figure 1. Automated log analysis.

The technique presented in this paper works in three phases, as shown in Figure 1. The *monitoring* phase consists of collecting logs from successful executions, at testing time or directly in-the-field. There exist several technologies that support generation of logs and can be used both during system development [2] and after systems have been released [16, 20]. Our approach does not refer to any specific monitoring solution and can be applied to any system independently from the logging technology.

In the *model generation* phase, the technique performs three major tasks: event detection, data transformation and model inference. Figure 2 shows the components involved in the model generation phase. In the next sections, we provide insights about this phase.

In the *failure analysis* phase, when a failure is observed, the corresponding log is retrieved and compared with the in-

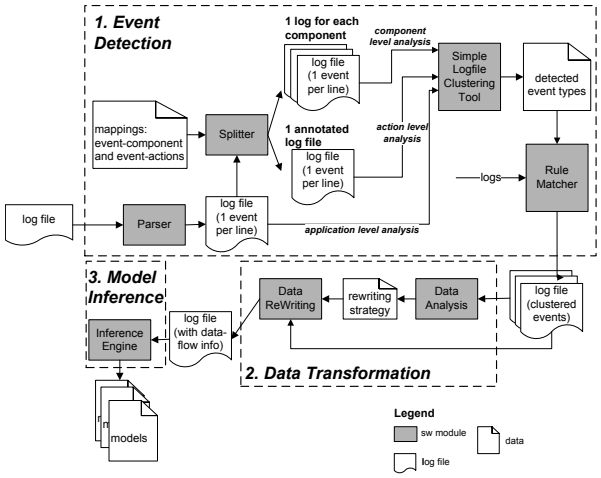


Figure 2. Model generation.

ferred models. The comparison algorithm identifies the accepted and the unaccepted event subsequences. The latter, namely suspicious subsequences, are presented to testers together with a representation of the behaviors that are acceptable instead of them, according to the inferred models. Testers use these inputs to identify the location and cause of failures.

These three phases may occur at different stages, e.g., monitoring at testing time, model inference after a system has been released and failure analysis when a problem is detected.

3 Event Detection

The event detection task is the first task executed in the model generation phase, as shown in Figure 2. The objective of this task is to rewrite the events in the initial log file, where events and their attributes are recorded as strings without a known structure, into a sequence of events where attribute values are separated from event names. We achieve this result in four steps.

In the first step, a parser is used to read the initial log file and produce a new version of the log file where an event and its attributes are stored in a single line. The parser requires the specification of the character that must be interpreted as an event separator. If the initial log file already satisfies this requirement, this step can be skipped.

In the second step, the splitter refines the log file according to the granularity selected for the analysis. Our technique supports three granularity levels: application, components and user actions. The application granularity produces a single log file that includes all the events generated by an application. This strategy is always applicable and does not require the intervention of the splitter. The component granularity produces one log file for each component of the system and requires the possibility to distinguish which components generated which events. The splitter implements this strategy by accepting as input a set of regular expressions that specify the events related to each component, and dividing the target log file into multiple log files. Each file includes all and only the events related to a component of the system. The user action granularity produces one log file for each user action and requires both the presence of a reactive system and the possibility to distinguish which events have been generated in response to which user input. The splitter supports this strategy by accepting as input a specification of the events that indicate the beginning and the termination of the execution of each user action, and dividing the input log file into multiple log files. Each file includes all and only the events related to an user action.

The use of a single log file for the whole application is the simplest strategy and allows to identify the causes of failures related to the interleaving of events generated by multiple components of the system. However, the use of a single log file for the whole application results in a large model that is sometime hard to process. On the other hand, the use of log files focusing on events generated by single components or single user actions reduces the size of the execution space and results in models that precisely represent the events generated by the execution of components and user actions, respectively. These strategies are effective in identifying anomalous event sequences related to single components or single user actions, but are not effective in identifying problems related to the interleaving of events generated by multiple components or multiple user actions.

In the third step of this task, we separate the event names from their attributes. To eliminate manual user intervention, we automatically perform this operation by using the Simple Logfile Clustering Tool (SLCT) [32]. This tool identifies prefixes that frequently appear in logged events and generate a set of regular expressions that specify how to separate the constant part, i.e., the event name, from the variable part, i.e., its attributes. For instance, if a log file includes both the events `login ''leonardo'' ''pwd''` and `login ''guest''`, SLCT can produce the regular expression `login *` to indicate that `login` should be considered as the event name and the rest should be considered as attribute values.

SLCT works by identifying the words that appear in each line, grouping lines with common words in clusters, and generating regular expressions that well represent all the elements in a cluster. SLCT discards incidental regular expressions by computing the support associated with each regular expression, and discarding the ones with a support below a given threshold. The support of a regular expression is the number of events in the analyzed log file that are generated by the regular expression.

In some cases, regular expressions can incidentally include some attribute values as part of the event name. This happens when the same specific attribute values are frequently present in the input log file. For instance, if the users ``leonardo'' and ``guest'' are the only users who log into a system, SLCT can identify the two regular expressions `login ``guest''` and `login ``leonardo'' *`, instead of `login *`. When there are multiple options in the regular expressions that can be generated, SLCT gives priority to the ones with the highest number of constant values, i.e., `login 'guest'''` and `login ``leonardo'' *` have priority on `login *`.

To avoid imprecise detection of event names, we identify regular expressions in multiple iterations. In the first iteration, we identify regular expressions with a support that is greater than 5% of the number of events recorded in the log file. If no regular expressions are identified, we reduce the requested support by 25% and we run again the algorithm. If regular expressions are identified, we consider the events generated by no regular expressions and we execute again the algorithm with this set. We continue in this way until the threshold for the support is less or equal to 1, or no events need to be analyzed. This process ease the identification of event names because general rules, such as the one containing only the event name, can more easily satisfy higher thresholds, than the specific rules with attribute values incidentally included as part of the event name.

In the fourth step, the rule matcher uses the regular expressions obtained by SLCT to generate a log file where event names are separated from attribute values. To this end, the rule matcher executes two operations: it assigns a symbolic identifier to each rule and it rewrites the input log file by replacing each constant part of the rule, i.e., the event name, with its identifier. For instance, if the rule `login *` is assigned to identifier `A`, the event `login ``leonardo'' ``pwd''` would be rewritten as `A ``leonardo'' ``pwd''` and the event `login ``guest''` would be rewritten as `A ``guest''`. The rewritten log files is the result of the event detection task.

4 Data Transformation

The data transformation task is the second task executed in the model generation phase, as shown in Figure 2. The goal of this task is to abstract from concrete attribute values, and replace them with data flow information that captures the rationale underlying the use of these values.

It is necessary to abstract from concrete values because they are data usually too specific to the executions in which they have been observed, and cannot be directly compared with other executions. For instance, consider the sequence of events <START T0, START T1, WAIT T0 10233621, WAITED T0 10233621, WAIT T1 29919449, WAITED T1 29919449, NOTIFIED T0 10233621, WAIT T0 10656878, WAITED T0 10656878> and the sequence <START T0, START T1, WAIT T0 11456633, WAITED T0 11456633, WAIT T1 33194956 WAITED T1 33194956>. Let us assume that the first sequence corresponds to a legal execution, the second sequence corresponds to a system failure, and we want to compare the two sequences to discover suspicious events[1]. If we compare both the event names and the event data, we would identify the sequence <WAIT T0 11456633, WAITED T0 11456633, WAIT T1 33194956 WAITED T1 33194956> as suspicious. However, the suspicious sequence shows that threads are waited exactly in the same order they are started, as in the legal sequence. It is clear that if we consider differences on data values as relevant, there is a high probability to generate false alarms (also known as false positives). A technique that produces many false alarms would be of little practical use because testers would spend most of their time in inspecting non-existing problems.

A simple alternative is ignoring data values. However, completely ignoring data values can miss many important suspicious event sequences. For instance, if the sequence collected in a failing execution is <START T0, START T1, WAIT T1 33194956 WAITED T1 33194956, WAIT T0 11456633, WAITED T0 11456633>, and we compare only event names without considering data values, we would not discover anything suspicious. However, attributes carry on an important information: the order of waited threads is changed. This difference can be relevant in the identification of possible sources of failures, e.g., deadlocks.

We can conclude that both solutions are not effective: directly comparing data values and event names can lead to many false alarms, while comparing event names only can miss many important suspicious events.

We deal with concrete values associated with events by replacing them with symbols that capture data flow information. We defined three rewriting strategies that can capture different data flow information: global ordering, relative to instantiation and relative to access.

Rewriting strategies are applied to group of homogeneous attributes, i.e., attributes that work on the same data. The application of the rewriting strategies to group of homogeneous attributes reduces the possibility to incidentally correlate unrelated data, thus decreasing the probability to generate false alarms.

Each group of attributes that should be targeted by a same rewriting strategy is identified by comparing attribute values, and heuristically assuming the existence of a correlation between the attributes that share a relevant number of values. In particular, if two attributes share more than 70% of their values (this parameter can be configured), they are

[1] to simplify the example we compare two sequences instead of comparing a model and a sequence

in a same group. The sets of attributes to be target by a same rewriting process are obtained by computing the transitive closure of the correlated attributes (we name one of these sets a data-flow cluster). In the rest of this section, the examples rewriting strategies are assumed to be applied within a same data-flow cluster.

Rewriting strategies are implemented by the Data Rewriting component shown in Figure 2. The Data Analysis component is used to automatically select the rewriting strategy to be applied according to the nature of the analyzed log file. The description of the Data Analysis component follows the description of the rewriting strategies.

Global Ordering The simplest way to remove the dependency from concrete values is consistently replacing concrete values with numbers, respecting the order of appearance. Thus, the first concrete value is rewritten as 1, the second concrete value is rewritten as 2 if never observed before, otherwise the same number is consistently used, and so on using sequential integer values. For instance, given two sequences <START T0, START T1, WAIT T0 10233621, WAITED T0 10233621, WAIT T1 29919449, WAITED T1 29919449> and <START T0, START T1, WAIT T0 11456633, WAITED T0 11456633, WAIT T1 33194956, WAITED T1 33194956> both of them would be rewritten as <START 1, START 2, WAIT 1 3, WAITED 1 3, WAIT 2 4, WAITED 2 4>.

Relative to Instantiation Some interesting patterns consist of: generate new values, use these values, then introduce further new values, use these values, and so on for many iterations. For instance, the sequence <START T1, START T2, STOP T1, DO 3, START T4, START T5, STOP T4, ...> shows a repeated schema where two new values are introduced and then the first one is used: T1 and T2 are introduced and then T1 used, later on T4 and T5 are introduced and T4 used.

To capture such patterns, we rewrite concrete values according to a strategy that focuses on the generation and use of values. In particular, each time a new value is generated, it is rewritten as a 0. If an existing value is detected, it is replaced with a number that indicates the number of new values that have been introduced from its first occurrence plus 1. Thus, the example sequence above would be rewritten as <START 0, START 0, STOP 2, DO 0, START 0, START 0, STOP 2, ...>, which well represents the detected repeated behavior. Note that this sequence would not be captured by the first strategy.

Relative to Access Relative to instantiation is useful when new concrete values are generated and then used, but does not work well when concrete values are reused multiple times. For instance, if we consider the sequence <START T0, START T1, WAIT T0, WAIT T1, STOP T0, STOP T1, START T2, START T0, WAIT T2, WAIT T0, STOP T2, STOP T0>, the previous rewriting strategies would result in <START 0, START 0, WAIT 2, WAIT 1, STOP 2, STOP 1, START 0, START 3, WAIT 1, WAIT 3, STOP 1, STOP 3>. We can notice that the initial sequence includes a repeated pattern with two threads being started, waited, and then stopped. This pattern is not captured by the relative to initialization strategy because thread names are reused multiple times.

To capture these patterns, we defined the relative to access rewriting strategy that replaces a concrete value with 0 if it is its first occurrence, otherwise it replaces it with a number that indicates the number of events observed from its last occurrence. For instance, the example sequence above would be rewritten as <START 0, START 0, WAIT 2, WAIT 2, STOP 2, STOP 2, START 0, START 3, WAIT 2, WAIT 2, STOP 2, STOP 2> that well represents two threads that are started, then waited and finally stopped in the same order.

Data Analysis Our solution includes three rewriting strategies. Each strategy focuses on different aspects, and it is hard to say a-priori which one should be applied. The rationale for choosing a strategy mainly depends on the nature of the log file. Since log files can be extremely large, testers can hardly manually inspect them to choose the proper rewriting strategy. Thus, we developed an automated technique that analyzes a log file and identifies the rewriting strategy to be applied to each data-flow cluster. This technique is implemented by the Data Analysis component shown in Figure 2.

The rationale for choosing the rewriting strategy is based on the observation that a strategy that applies well to a particular sequence of values would capture its regularity, thus using a small quantity of distinct symbols to represent it. Global ordering is effective when a same set of values is reused several times, relative to instantiation is effective when new values are created and then used few times, relative to access is effective when values are created and then reused multiple times. To automatically select the rewriting strategy to be used for a given data-flow cluster, we apply all the three techniques to the cluster and then we select the one that produces the smallest number of symbols. Since spurious values can be present in data-flow sequences and cause the generation of extra symbols, we select the technique that rewrites more than 50% of concrete values with the less number of symbols.

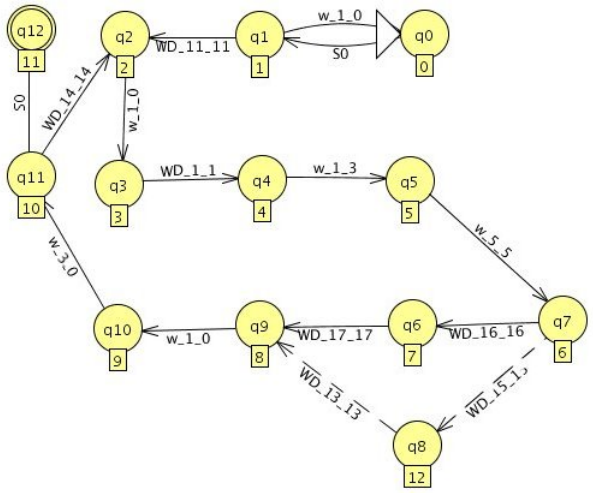
In some cases, data-flow clusters can include attribute values not distributed according to any of our strategies. In that cases, even the best rewriting strategy can provide poor results and cause several false positives in the final analysis. To avoid this issue, we select a rewriting strategy for a data-

flow cluster only if attribute values can be rewritten by using at most 10 symbols. If more symbols are necessary for a data-flow cluster, we simply delete attribute values and we work by considering event names only.

Rewriting strategies are very simple and the time required by data analysis is minimal. However, in case of huge log files, the time for this analysis can be reduced by only analyzing a sample subset of the log file.

5 Model Inference

In the model generation task, we apply the kBehavior inference engine [23] on log files with rewritten attributes. KBehavior incrementally analyzes the preprocessed log files and generates a FSA that both summarizes and generalizes the observed event sequences. At each step, kBehavior reads a trace and updates the current FSA according to the content of the trace. The updated FSA guarantees to generate all the traces that have been analyzed.



input trace: `S0, WD_11_11, w_1_0, WD_1_1, w_1_3, w_5_5, WD_15_15, WD_13_13, w_1_0, w_3_0, S0`

Figure 3. An example FSA extended with a new trace. The state with the triangle is the initial state. The state with the double border is the final state. The dotted arrows and state `q8` are added by the extension step.

The algorithm used by kBehavior to extend a current FSA given a new trace is based on the identification of submachines in the current FSA that generate sub-sequences in the input trace. Once these relations are identified, the portions of the input trace that do not correspond to any submachine are used to create new branches in the current FSA, so that the updated FSA generates both all event sequences generated by the previous FSA and the event sequences represented in the input trace. For example, Figure 3 shows how a FSA can be extended providing a new input string to kBehavior. In this simple example, the portion of the input string that does not correspond to any submachine lead to the addition of a new branch to the current FSA. In the general case, a FSA is extended by gluing the FSAs obtained from the (recursive) execution of kBehavior on the portions of the input string that cannot be associated with any submachine. Technical details can be found in [23].

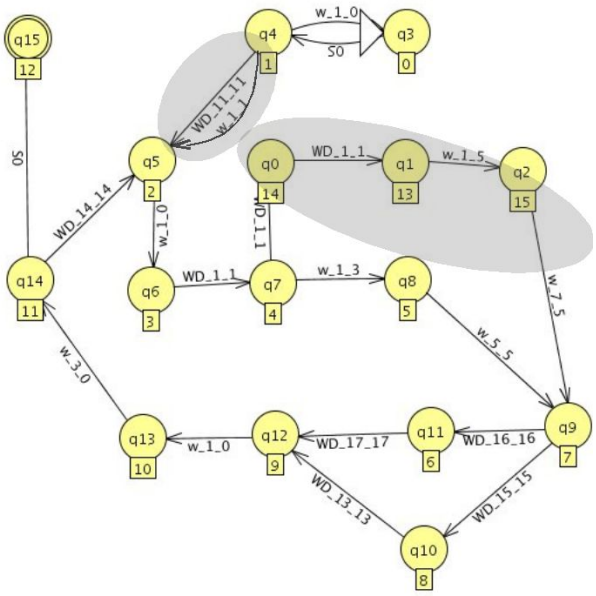
If testers selected the application level granularity, the model inference task produces one single model for the whole application. If testers selected the component or user action granularity levels, the model inference task produces one model per component or per user action, respectively.

6 Failure Analysis

In the failure analysis phase, we compare the behavior described by the inferred models with log files recorded during failing runs. The goal of the comparison is to automatically identify anomalous patterns, i.e., event sequences that differ from the ones observed in previous executions. Anomalous patterns are likely to indicate the source and the cause of the failure, and can be efficiently inspected by testers independently from the original size of the log file.

A straightforward way to compare a trace and a model is to check if the model generates the trace. If the trace is generated by the model, there are no anomalies. If the trace is generated only up to a given point, namely p, there is an anomaly. The tester can thus inspect the set of events around p and the sub-machine around the state that has been reached by generating all the symbols of the trace up to position p. Unfortunately, this strategy has little effectiveness when multiple anomalies or noisy data is present in the analyzed trace. If a trace includes multiple anomalous patterns, the first anomaly in the trace hides all successive anomalies. This happen because when an anomaly is detected, the remaining portion of the trace cannot be matched anymore and the checking is interrupted.

To avoid loss of important information, we implemented the matching process between traces and FSAs on top of the kBehavior extension mechanism, which is extremely useful in pairing event sequences and sub-machines independently from their positions. For instance, a sub-sequence that is located at the beginning of the trace can be associated with any sub-machine of the FSA. Thus, we use the trace to be matched to extend the current model, and we consider all the extensions points as the set of anomalous events that must be inspected by testers. Note that many extensions points of arbitrary length and complexity can be identified, thus making the approach extremely flexible and powerful,



trace to be matched: `S0, w_1_1, w_1_0, WD_1_1, WD_1_1, WD_1_1, w_1_5, w_7_5, WD_16_16, WD_17_17, w_1_0, w_3_0, S0`

Figure 4. An example of a matching between a FSA and a trace. The gray areas indicate the parts that should be added as a consequence of an extension step. In this case, they represent the anomalies that are presented to testers.

even for the identification of multiple anomalies located in different points of the model. In fact, presence of noisy data or multiple anomalies do not hinder effectiveness of the matching process because all anomalous sequences are identified. Figure 4 shows an example of a matching between a trace and a FSA.

7 Empirical Work

Since the final output of our technique is a set of suspicious sequences that testers investigate to identify failure causes, the evaluation of the effectiveness of the technique is based on the analysis of failure causes on a set of case studies. To discuss the quality of the results, we computed the number of true and false positives, and we verified the capability of testers to quickly point to failure causes. Since the technique supports different kinds of granularity (application, component and user action levels), we compared the results obtained in the different cases.

Table 1 summarizes the considered case studies: CASS, a commercial air traffic control simulator developed by Artisys [4]; Glassfish, which is a J2EE application server (2.000.000 lines of code) [9] and Tomcat, which is a JSP/Servlet server (300.000 lines of code) [3]. We evaluated our technique against a set of known faults affecting these systems: an algorithmic fault manually injected in CASS, two Glassfish configuration problems [11, 12], and two faults affecting Glassfish and Tomcat [10, 30].

For each case study, the empirical investigation followed three steps: (1) we derived test cases for target systems by using the category partition method [25] and we collected log files from successful executions, (2) we reproduced the failures and we collected the corresponding log files, and (3) we generated models from the log files recorded for successful executions and we used these models to identify the causes of the failures. We collected log files using the highest verbosity with the exception of G2, where the default verbosity has been used.

Results are summarized in Table 2. We run our technique with the three granularity levels for all the case studies with the exception of A2, where the user action granularity level is not applicable. For each case study we indicate: the percentage of suspicious events, the number of false positives, i.e., the number of suspicious events not related to the investigated failures, the number of true positives, i.e., the number of suspicious events related to the investigated failure, and the precision of the results.

The percentage of events identified as suspicious by our technique gives an indication of the effort required to debug faults. In fact, the user has to manually inspect each set of suspicious events.

In all the case studies, our technique presented to testers a small set of suspicious sequences to be investigated (from 0.03% in the best case, to 28.57% in the worst case), and the suspicious events always included an explanation to the investigated failure. In G2 our technique identified an exception caused by a configuration problem; in G3 and T1, it identified faulty class initializations and faulty load events; finally in A1, it detected a wrong sequence of values that caused the system failure. All these issues are related to unexpected combinations of events and attribute values. The high reduction in the number of events to be inspected results in an important save of effort and time by testers.

Data report a moderately high number of false positives. They are caused by incompleteness of the samples used for model inference and limited generalization in the inference process. For instance, if some events in log files always appear with same attribute values, SLCT can imprecisely partition event names from attributes. Similarly, a failure that executes a part of the system that has been never executed before would generate several false alarms.

Even if some false positives are often present, the technique still results in a important advantage for testers, who need to analyze a small percentage of the faulty logs to

	Case study	Failure type	Log format	Log size
A1	Air traffic control simulator	Wrong system behavior	syslog	1Mb
G1	GlassFish (version 2-GA)	Petstore not deployed [11]	uniform log	1Mb
G2	GlassFish (version 2-GA)	Petstore not deployed [12]	uniform log	84Mb
G3	GlassFish (version 3-b01)	Server hangs [10]	uniform log	47Mb
T1	Tomcat (version 6.0.4)	Web application not started [30]	java simple log	17Mb

Table 1. Case studies

Analysis	Component Level Analysis				Action Level Analysis				Application Level Analysis			
Case study	Reported events	FP	TP	Precision	Reported events	FP	TP	Precision	Reported events	FP	TP	Precision
A1	1.88%	9	3	0.25	-	-	-	-	1.73%	5	7	0.64
G1	28.57%	3	5	0.63	28.25%	0	5	1.00	3.57%	1	0	0.00
G2	0.03%	17	10	0.37	0.05%	37	8	0.18	0.05%	34	7	0.17
G3	0.04%	44	13	0.16	0.07%	79	10	0.11	0.03%	39	7	0.15
T1	1.07%	6	12	0.67	1.25%	16	5	0.24	2.79%	4	8	0.67

Table 2. Results obtained with different granularity levels (see legend in Table 4).

diagnose failure causes. Furthermore, we experimentally detected that testers can reduce the number of experienced false positives by restricting the analysis to the events generated in failing user actions. If the events that indicate beginning and termination of user actions are defined, this reduction can be always applied, independently from the granularity level selected for the analysis. It is worth to mention that this optimization can be applied only to reactive systems, where user actions can be defined. Table 3 summarizes the results when this reduction is applied. In the majority of the cases, precision improves substantially at the cost of losing some true positives. This is due to the restricted scope of the optimized analysis that may miss some relevant anomalies that are outside the scope of the failing user action.

The three granularity levels provided different results in the five case studies. In cases T1 and G2, component level analysis performs better than the others. In these two cases logging recorded a high number of events. Since component level analysis focuses on one component at time, the derived models are more compact and precise than the ones derived with the other two approaches. In case G1, the action level analysis works better than the others. In this case the number of events recorded in log files are limited, thus allowing the generation of precise models that describe the events generated in response of user actions. Finally, in case A1, the application level analysis provided the best results. In this case, most of the false positives have been generated by a part of the system that has been never executed in the monitoring phase. The application granularity collapsed all the new events in a single extension of the model, which can easily and quickly be identified as a false alarm. Models generated with the component level granularity, instead, produced several false positives, because the failing execution traversed many functionalities never monitored before, thus resulting in a high number of model violations.

The component level granularity in average performs better than the others and can be applied even when a rich set of events and attributes is logged. User action granularity scales worse than component level because many events and attributes can be observed when a user action is executed, but it is effective in detecting the causes of failures when they are concentrated in a single user action. Finally, application level granularity is the one which provides the best results when few events and attributes are logged per execution, but suffers of scalability problems when large systems or huge log files are considered.

Table 4 shows data about both true positives identified thanks to the analysis of data values and information about size of the inferred and analyzed models, for the configurations that provided the best results. The relevant number of true positives presented to testers thanks to the analysis of data flow relations demonstrate the importance of working on both events and attribute values, which is one of the distinguishing characteristics of our solution. Data about the size of models indicate the capability of our technique to both handle and derives models of non-trivial size for real systems.

8 Related Work

Log file analysis is commonly used to diagnose and investigate failures that have been experienced either in the field or during testing. We can classify log file analysis approaches in three classes: specification-based techniques, expert systems and heuristic based techniques.

Analysis	Component Level Analysis				Action Level Analysis				Application Level Analysis			
Case study	Reported events	FP	TP	Precision	Reported events	FP	TP	Precision	Reported events	FP	TP	Precision
A1	1.88%	9	3	0.25	-	-	-	-	1.73%	5	7	0.64
G1	28.57%	3	5	0.63	28.25%	0	5	1.00	3.57%	1	0	0.00
G2	0.01%	2	6	0.75	0.02%	9	6	0.35	0.01%	7	4	0.36
G3	0.03%	29	13	0.31	0.04%	43	7	0.14	0.02%	23	7	0.23
T1	0.71%	4	8	0.67	0.65%	5	6	0.55	1.90%	8	14	0.44

Table 3. Results obtained by restricting the analysis to the failing user action (see legend in Table 4).

		DF data relevance		Models size			
Case study	Granularity	TP	TP using DF data	num FSA	avg states	avg trans	avg symbols
A1	application	7	7 (100%)	1	338	373	117
G1	action	5	0 (0%)	2	84	130	74
G2	component	10	2 (20%)	67	23	56	19
G3	component	13	2 (15%)	41	8	27	9
T1	component	11	7 (64%)	37	32	76	47

Legend:

`FP`	false positives
`TP`	true positives
`TP using df data`	TP detected by using data flow information
`num FSA`	number of FSA
`avg states` `avg trans` `avg symbols`	average number of states, transitions, and symbols per FSA

Table 4. Results about relevance of attribute values in the analysis and size of the inferred models

Specification based approaches compare events recorded in log files with models that represent valid event sequences to detect failures and help developers in identifying their causes [1]. Experimental results demonstrate their effectiveness, but effort required to produce and maintain specifications reduces applicability of these techniques.

Expert systems, like Logsurfer [19], SEC [31] and Swatch [14], compare events recorded in log files with a set of event patterns, usually specified with regular expressions, that are known to correspond to system failures. Thus, effectiveness of these techniques are limited by the set of known problems and available patterns. Moreover, issues related to maintenance of specifications also apply to maintenance of event patterns.

The technique presented in this paper overcomes the issues related to availability and maintenance of specifications and event patterns by automatically inferring models of the legal behavior of software systems, and using these models to detect failure causes.

Other work heuristically addressed lack of specifications by applying supervised and unsupervised learning to log files, similarly to our approach. In the case of supervised learning, user intervention is required in the learning phase, while in unsupervised learning the learning is fully automated. Learning techniques can derive different models that represent the legal behaviors recorded in log files, e.g., decision trees [6], statistical models [5] and clusters [32, 27]. These techniques focus on problems due to single unexpected events, while we address the more general case of multiple unexpected events.

Finally, a body of related work comes from the security field, where unexpected events detected in log files are interpreted as attempts to violate systems. These techniques monitor sequences of system calls, and detect subsequences never observed in valid executions [15] or violations of temporal constraints [21]. Such approaches can recognize problems depending on complex event sequences, but cannot capture unexpected data flows.

9 Conclusions

Runtime data is frequently collected from systems running in the field for the purpose of debugging. In fact, when a failure is observed in the field, analysis of log files is typically the first activity that is performed. Unfortunately, manual analysis of log files is seldom practical, and testers need to be supported by automated techniques that indicate the suspicious event sequences that deserve further investigations.

Existing solutions for automated log file analysis are limited in many aspects. Some techniques require formal specifications to be applied, but formal specifications are seldom available in practice [1]. Other techniques work well for problems caused by single unexpected events, but do not address failures caused by a combination of multiple events [6, 32].

In this paper, we presented a technique for automated log file analysis that can identify problems caused by mul-

tiple unexpected event sequences and data flows, and requires minimal user intervention. This work develops several ideas: it defines strategies to handle concrete data values and automatically discover data-flow information; it derives models that support the discovery of precise illegal event sequences juxtaposed to expected legal sequences; it automatically identifies clusters of events and clusters of attributes to automate problem abstraction and reduction.

A set of empirical experiences with large systems show the effectiveness of the approach. In particular, the technique identified the suspicious sequences and the anomalous data values responsible for five investigated problems.

Further studies are needed to generalizes results about effectiveness of the proposed solution. Even if we can capture types of problems not well addressed by other techniques, we aim at running a set of comparative case studies for strengthening this result. Moreover, further controlled experiments will be executed to derive data about false negatives, i.e., issues in the log files missed by our technique, that are now not well addressed in our experiments.

Acknowledgment This work has been 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] J. H. Andrews and Y. Zhang. Broad-spectrum studies of log file analysis. In *Proceedings of the IEEE International Conference on Software Engineering*, 2000.

[2] Apache Software Foundation. Log4java. http://logging.apache.org/log4j/, visited in 2008.

[3] Apache Software Foundation. Tomcat JSP/Servlet server. http://tomcat.apache.org/, visited in 2008.

[4] Artisys. Cass, air traffic control simulator. http://www.artisys.aero/, visited in 2008.

[5] P. Bodic, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. I. Jordan, and D. Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Proceedings of the 2nd IEEE International Conference on Automatic Computing*, 2005.

[6] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Proceedings of the International Conference on Autonomic Computing*, 2004.

[7] S. K. Chilukuri and K. Doraisamy. Symptom database builder for autonomic computing. In *Proceedings of the IEEE International Conference on Autonomic and Autonomous Systems*, 2006.

[8] D. Cotroneo, R. Pietrantuono, L. Mariani, and F. Pastore. Investigation of failure causes in workload-driven reliability testing. In *Proceedings of the ACM Fourth International Workshop on Software Quality Assurance*, 2007.

[9] Glassfish application server. Glassfish. https://glassfish.dev.java.net/, visited in 2008.

[10] Glassfish JIRA bug database. Glassfish issue 4255. https://glassfish.dev.java.net/issues/showbug.cgi?id=4255, visited in 2008.

[11] Glassfish user forum. Glassfish configuration issue. http://forums.java.net/jive/thread.jspa?messageID=252898, visited in 2008.

[12] Glassfish user forum. Glassfish configuration issue. http://forum.java.sun.com/thread.jspa?threadID=5249570, visited in 2008.

[13] A. Gordon. *Classification*. Chapman and Hall/CRC, 1999.

[14] S. E. Hansen and E. T. Atkins. Automated system monitoring and notification with swatch. In *Proceedings of the 7th USENIX conference on System administration*, 1993.

[15] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[16] IBM. Eclipse test & performance tools platform. http://www.eclipse.org/tptp/, visited in 2007.

[17] P. Jackson. *Introduction to expert systems*. Addison Wesley Longman, 1999.

[18] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. Wiley, 1990.

[19] Kerry Thomson. Logsurfer log monitoring tool. http://www.crypt.gen.nz/logsurfer/, visited in 2008.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-oriented Programming*, 2001.

[21] W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from unix process execution traces for intrusion detection. In *Proceedings of the AAAI97 workshop on AI Approaches to Fraud Detection and Risk Management*, 1997.

[22] C. Lonvick. The bsd syslog protocol, 2001.

[23] L. Mariani and M. Pezzè. Dynamic detection of COTS components incompatibility. *IEEE Software*, 24(5):76–85, 2007.

[24] C. L. Mendes and D. Reed. Monitoring large systems via statistical sampling. *International Journal of High Performance Computing Applications*, 18(2):267–277, 2004.

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

[26] J. Prewett. Analyzing cluster log files using logsurfer. In *4th Annual Conference on Linux Clusters*, 2003.

[27] J. Stearley. Towards informatic analysis of syslogs. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, 2004.

[28] Sun. Glassfish v3 application server administration guide. http://docs.sun.com/doc/820-4495, visited in 2008.

[29] Sun. Java logging overview. http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.htm l, visited in 2008.

[30] Tomcat bugzilla bug database. Tomcat issue 40820. https://issues.apache.org/bugzilla/showbug.cgi?id=40820, visited in 2008.

[31] R. Vaarandi. Sec - a lightweight event correlation tool. In *Proceedings of the 2002 IEEE Workshop on IP Operations and Management*, 2002.

[32] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations and Management*, 2003.