Identifying Incompatible Service Implementations using Pooled Decision Trees

Christian Inzinger, Waldemar Hummer, Benjamin Satzger,
Philipp Leitner and Schahram Dustdar
Distributed Systems Group, Vienna University of Technology
Argentinierstrasse 8, A-1040 Vienna, Austria
{lastname}@infosys.tuwien.ac.at

ABSTRACT

We study fault localization techniques for identification of incompatible configurations and implementations in service-based applications (SBAs). Practice has shown that standardized interfaces alone do not guarantee compatibility of services originating from different partners. Hence, dynamic runtime instantiations of such SBAs pose a great challenge to reliability and dependability. The aim of this work is to monitor and analyze successful and faulty executions in SBAs, in order to detect incompatible configurations at runtime. We propose an approach using pooled decision trees for localization of faulty service parameter and binding configurations, explicitly addressing transient and changing fault conditions. The presented fault localization technique works on a per-request basis and is able to take individual service inputs into account. Considering not only the service configuration but also the service input data as parameters for the fault localization algorithm increases the computational complexity by an order of magnitude. Hence, our performance evaluation is targeted at large-scale SBAs and illustrates the feasibility and decent scalability of the approach.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; I.2.6 [**Learning**]: Induction

General Terms

Algorithms, Experimentation, Management, Reliability

Keywords

Fault Localization, Service-Oriented Architecture, Pooled Decision Trees, Dependability

1. INTRODUCTION

Distributed and mission-critical enterprise applications are becoming more and more reliant on external services, provided by suppliers, customers or other members of service value networks [3] (SVNs). In many industries, the technical interfaces of these services are governed by industry standards, specified by bodies such as the TM Forum[1] (TMF), the Association for Retail Technology Standards[2] (ARTS) or the International Air Transport Association[3] (IATA). Hence, integration of services provided by different partners into a single service-based application (SBA) becomes feasible. Additionally, as oftentimes a multitude of potential partners are providing implementations of the same standardized interfaces, SBAs are enabled to dynamically switch providers at runtime, i.e., select the most suitable implementation of a given standardized interface based on current requirements.

Unfortunately, practice has shown that standardized interfaces alone do not guarantee compatibility of services originating from different partners. Many industry standards are prone to underspecification, while others simply allow multiple alternative (and incompatible) implementations to co-exist. Additionally, and particularly for younger specifications, not every vendor can be trusted to interpret each standard text in the same way. Consequently, there are practical cases where SBAs, which should work correctly in theory, fail to function because of unexpected incompatibilities of service implementations chosen at runtime. Note that this does not necessarily mean that any single one of the chosen service implementations is faulty in itself – it merely means that two or more chosen service implementations do not work in conjunction (even though both may work perfectly in combination with other services).

In this paper, we present a machine learning driven approach to identify such incompatibilities of industry standard implementations. We analyze runtime event logs emitted by the SBA using decision tree techniques and principal component analysis, with the goal of suggesting combinations of service implementations that should not be used in conjunction. Decision trees are a white-box machine learning approach that allow to extract incompatibility rules from the constructed tree [28]. Our approach takes into account not only the actual service implementations themselves, but also the received input and the produced output data of implementations. Furthermore, we quantify the benefits of our approach based on a numerical evaluation.

The remainder of the paper is structured as follows. The core part is presented is Section 2, where we establish a model for fault localization in SBAs and describe our ap-

SAC'13 March 18–22, 2013, Coimbra, Portugal

[1] http://www.tmforum.org/browse.aspx
[2] http://www.nrf-arts.org/
[3] http://www.iata.org/Pages/default.aspx

proach in detail. In Section 3 we briefly discuss the prototypical implementation of the system. Section 4 contains a comprehensive experimental evaluation and discusses strengths and limitations of the approach. Section 5 discusses related work in the field of reliable distributed systems and fault localization in SBAs. Finally, Section 6 concludes the paper and points to future research directions.

2. FAULT LOCALIZATION APPROACH

This section discusses our novel fault localization technique. Section 2.1 establishes a notion for the model of SBAs. Sections 2.2 and 2.3 discuss preprocessing and machine learning techniques used to learn rules which describe the reasons for faults based on the collected model data.

2.1 System Model

We establish a generalized model that forms the basis for the concepts presented in the paper. The core model artifacts are discussed in the following.

A SBA consists of a set of industry standard service interfaces $I = \{i_1, \ldots, i_n\}$ and a set of implementations $C = \{c_1, \ldots, c_m\}$. The mapping between interface and implementation is defined by the function $c : I \rightarrow \mathcal{P}(C)$, where $\mathcal{P}(C)$ denotes the power set of C. The domain of possible input parameters P, each defined by name (N) and domain of possible data values (D) is represented by $P = N \times D$. Function $p : I \rightarrow \mathcal{P}(P)$ returns all inputs required by an interface, and $d : P \rightarrow D$ returns the value domain for a given parameter. The set $F \subseteq I \times I$ defines data flows as pairs of interfaces (i_x, i_y), where the output of i_x becomes the input of i_y. Transitive data flows spanning more than two services can be derived from F. Moreover, we define $T = \langle t_1, \ldots, t_k \rangle$ as the sequence of logged execution traces $t_x : K \rightarrow V$ in chronological order, mapping the set of keys $K = I \cup (I \times N)$ to values $V = C \cup D$; interfaces I map to implementations C, whereas parameter names $I \times N$ map to parameter domains D. Finally, the function $r : \{1, \ldots, k\} \rightarrow \{success, fault\}$ is used to express the result of a trace $t_x, x \in \{1, \ldots, k\}$, i.e., whether the trace represents a successful or failed execution of the SBA.

Summarizing the model, the core idea of our approach is to analyze log traces of SBA executions for fault localization. We consider two classes of properties as part of the traces: 1) runtime binding of interfaces to concrete implementations, 2) service input parameters, i.e., data provided by the user to the application as well as data flowing between services.

2.2 Trace Data Preparation

Table 1 lists an excerpt of six exemplary traces for an imaginative customer-oriented SBA. The table contains multiple rows which represent the traces $(t_1, \ldots, t_6)$; the columns contain the bindings for the service interfaces $(i_1, i_2, i_3, \ldots)$, the input parameter values $(t_x(\ldots))$, and the success result of the trace $(r(x))$. Two exemplary parameters for a customer service are in the Table: $t_x(i_1, 'custID')$ denotes the customer identifier provided to some interface i_1, and the parameter $t_x(i_2, 'premium')$ tells the service interface i_2 whether it is dealing with a regular customer or a high-paying premium customer.

We follow the typical machine learning terminology and denote the column titles as *attributes* and the rows starting from the second row as *instances*. The first attribute (t_x) is the instance identifier, and $r(x)$ is denoted *class attribute*.

$\mathbf{t_x}$	$\mathbf{i_1}$	$\mathbf{i_2}$	$\mathbf{i_3}$	..	$\mathbf{t_x(i_1, 'custID')}$	$\mathbf{t_x(i_2, 'premium')}$	..	$\mathbf{r(x)}$
$\mathbf{t_1}$	c_1	c_3	c_7	..	$'joe123'$	$false$	..	$success$
$\mathbf{t_2}$	c_2	c_4	c_6	..	$'aliceXY'$	$true$	..	$success$
$\mathbf{t_3}$	c_1	c_5	c_8	..	$'joe123'$	$false$	..	$fault$
$\mathbf{t_4}$	c_2	c_5	c_8	..	$'bob456'$	$true$	..	$success$
$\mathbf{t_5}$	c_2	c_4	c_7	..	$'aliceXY'$	$true$	..	$success$
$\mathbf{t_6}$	c_1	c_4	c_8	..	$'lindaABC'$	$false$	..	$fault$
..								

Table 1: Example Traces for Sample Application

The number of attributes and combinations of attribute values can grow very large. To estimate the number of possible traces for a medium sized application, consider an SBA using 10 interfaces ($|I| = 10$), 3 candidate implementations per interface ($|c(i_x)| = 3 \ \forall i_x \in I$), 3 input parameters per service ($|p(i_x)| = 3 \ \forall i_x \in I$), and 100 possible data values per parameters ($|d| = 100 \ \forall i_x \in I, (n, d) \in p(i_x)$). The total number of possible execution traces in this SBA is $3^{10} * 100^{3^{10}} = 5.9049 * 10^{64}$. Efficient localization of faults in such large problem spaces evidently poses a huge algorithmic challenge. Even more problematically, the problem space becomes infinite if the service parameters use non-finite data domains (e.g., *String*). The first step towards feasible fault analysis is to reduce the problem space to the most relevant information. We propose a two-step approach:

1. Identifying (ir)relevant attributes: The first manual preprocessing step is to decide, based on domain knowledge about the SBA, which attributes are relevant for fault localization. For instance, in an e-commerce scenario we can assume that a unique customer identifier ($custID$) does not have a direct influence on whether the execution succeeds or fails. Per default, all attributes are deemed relevant, but removing part of the attributes from the execution traces helps to reduce the search space.

2. Partitioning of data domains: Research on software testing and dependability has shown that faults in programs are often not solely incurred by a single input value, but usually depend on a range of values with common characteristics [30]. Partition testing strategies therefore divide the domain of values into multiple sub-domains and treat all values within a sub-domain as equal. As a simple example, consider a service parameter with type *Integer* (i.e., $\{-2^{31}, \ldots, +2^{31} - 1\}$), a valid partitioning would be to treat negative/positive values and zero as separate sub-domains: $\{\{-2^{31}, \ldots, -1\}, \{0\}, \{1, \ldots, +2^{31} - 1\}\}$. If explicit knowledge about suitable partitioning is available, input value domains can be partitioned manually as part of the preprocessing. However, efficient methods have been proposed to automatize this procedure (e.g., [6]).

2.3 Learning Rules from Decision Trees

Using the preprocessed trace data, we strive to identify the attribute values or combinations of attribute values that are likely responsible for faults in the application. For this purpose, we utilize decision trees [24], a popular technique in machine learning. It has the advantage that the decision making of the resulting trees can be easily comprehended; their knowledge can be distilled for the purpose of fault localization. Also, decision tree training with state of the art algorithms like C4.5 results in comparably fast learning speeds, compared to other machine learning approaches.

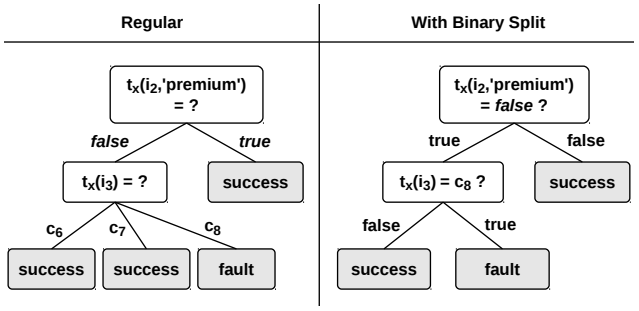

Figure 1: Exemplary Decision Tree in Two Variants

Figure 1 illustrates decision trees based on the example traces in Table 1. The figure shows two variants of the same tree which classifies non-premium services from Provider 3 ($t_x(i_3) = c_8$). The inner nodes are decision nodes which divide the traces search space, and the leaf nodes indicate the trace results. The left-hand side of the figure shows a regular decision tree where each decision node splits according to the possible values of an attribute. The right-hand side shows the same tree with binary split (i.e., each decision node has two outgoing edges).

Algorithm 1 Obtain Incompatibility Rules from Decision Tree

```
1: E_I ← ∅
2: for all fault leaf nodes as n do
3:    path ← path of nodes from n to root node
4:    E_temp ← ∅
5:    for all decision node along path as d do
6:       c ← condition of d
7:       if c is true along path then
8:          E_temp ← E_temp ∪ c
9:       end if
10:   end for
11:   E_I ← E_I ∪ E_temp
12: end for
13: for all E_x, E_y ∈ E_I do
14:   if E_x is covered by E_y then
15:      E_I ← E_I \ E_x
16:   end if
17: end for
```

The decision tree with binary split is used to automatically derive incompatible attribute values. The basic procedure is to loop over all $fault$ leaf nodes and create a combination of attribute assignments along the path from the leaf to the root node. The detailed algorithm is presented in Algorithm 1. For each $fault$ leaf node, a set E_{temp} is constructed which contains the conditions that are true along the path. The total set of all such condition combinations is denoted E_I. Our approach exploits the simple structure of decision trees for extracting incompatibility rules; other popular classification models (e.g., neural networks) have much more complex internal structures which make it harder to extract the principal attributes responsible for the output [28].

2.4 Coping with Transient Faults

So far, we have shown how trace data can be collected, transformed into a decision tree, and used for obtaining rules which describe which configurations have led to a fault. The assumption so far was that faults are deterministic and static. However, in real-life systems which are influenced by various external factors, we have to be able to cope with temporary and changing faults. Our approach is hence tailored to react to such irregularities in dynamically changing environments.

A temporary fault manifests itself in the log data as a trace $t \in T$ whose result $r(t)$ is supposed to be $success$, but the actual result is $r(t) = fault$. Such temporary faults can lead to a situation of contradicting instances in the data set. Two trace instances $t_1, t_2 \in T$ contradict each other if all attributes are equal except for the class attribute:
$\{(k, v) \mid (k, v) \in t_1\} = \{(k, v) \mid (k, v) \in t_2\}$, $r(t_1) \neq r(t_2)$.

Fortunately, state-of-the-art decision tree induction algorithms are able to cope with such temporary faults which are considered as noise in the training data (e.g., [1]). If the reasons for faults within an SBA change permanently, we need a mechanism to let the machine learning algorithms forget old traces and train new decision trees based on fresh data. Before discussing strategies for maintaining multiple decision trees, we first briefly discuss in Section 2.5 how the accuracy of an existing classification model is tested over time.

2.5 Assessing the Accuracy of Decision Trees

Let D be the set of decision trees used for obtaining fault combination rules. We use the function $rc : (D \times \{1, \ldots, k\}) \rightarrow \{success, fault\}$, where k is the highest trace index, to express how a desicion tree classifies a certain trace. Over a subset $T_d \subseteq T$ of the traces classified by a decision tree d, we assess its accuracy using established measures true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) [2].

From the four basic measures we obtain further metrics to assess the quality of a decision tree. The *precision* expresses how many of the traces identified as faults were actually faults ($TP/(TP + FP)$). *Recall* expresses how many of the faults were actually identified as such ($TP/(TP + FN)$). Finally, the *F1 score* [9] integrates precision and recall into a single value (harmonic mean):
$F1(d) = 2 * \frac{precision \cdot recall}{precision + recall}$

2.6 Maintaining a Pool of Decision Trees

In the following we discuss our approach to cope with changing fault conditions over time, based on a sample execution of the system model introduced in Section 2.1.

Figure 2 illustrates a representative sequence of execution traces ($\{t_1, t_2, t_3, \ldots\}$); time progresses from the left-hand side to the right-hand side of the figure. In the top of the figure the trace results ($r(t_x)$) are printed, where "S" represents $success$ and "F" represents $fault$. As the traces arrive with progressing time we utilize deduction algorithms to learn decision trees from the data. At time point 1, the decision tree d_1 is initialized and starts the training phase. The learning algorithm has an initial training phase which is required to collect a sufficient amount of data to generate rules that pass the required statistical confidence level. After the initial training phase the quality of the decision tree rules is assessed by classifying new incoming traces. In Figure 2 correct classifications are printed in normal text, while incorrect classifications are printed in bold underlined font.

We have marked four particularly interesting time points (a, b, c, d) in Figure 2, which we discuss in the following.

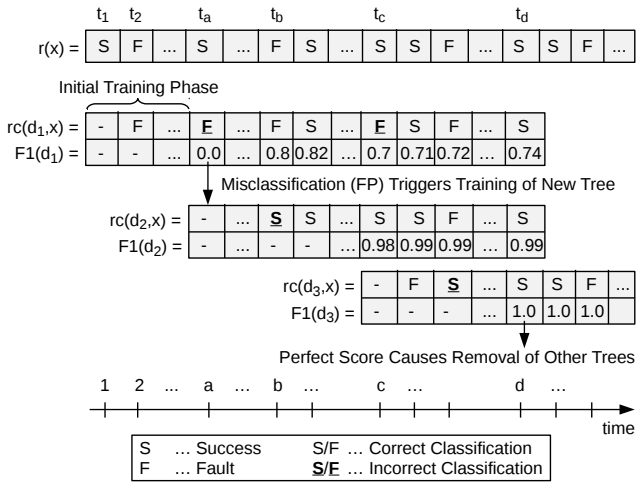

Figure 2: Maintaining Multiple Trees to Cope with Changing Faults

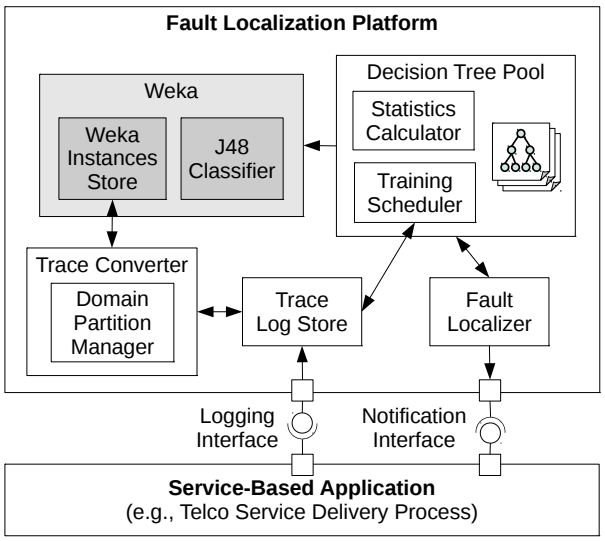

Figure 3: Prototype Implementation Architecture

1) At time a the tree d_1 misclassifies the trace t_a as a false positive. This triggers the parallel training of a new decision tree d_2 based on the traces starting with t_a. 2) A false negative by d_2 occurs at time b. However, since this happens during the initial training phase of d_2, we simply regard the trace t_b as useful information for the learner and add it to the training set. No further action is required. 3) Time point c contains another false positive misclassification of d_1. In the meantime, $F1(d_1)$ had risen due to some correct classifications, but now the score is pushed down to 0.7. Again, as in time point a, the generation of a new tree d_3 is triggered. 4) At time d the environment seems to have stabilized and decision tree d_3 reached a state with perfect classification ($F1(d_3) = 1$). At this point, the remaining decision trees are rejected. The old trees are still stored for reference, but are not trained with further data to save computing power.

3. IMPLEMENTATION

Our prototype implementation of the presented fault localization approach is implemented in Java. We utilize the open-source machine learning framework *Weka*[4]. Weka contains an implementation of the popular *C4.5* decision tree deduction algorithm [25], denoted *J48 classifier* in Weka. C4.5 has been applied successfully in many application areas and is known for its good performance characteristics.

Figure 3 outlines the architecture of the Fault Localization Platform with the core components. Third-party components (Weka) are depicted with light grey background color. The service-based application submits its log traces (service bindings plus input messages) to the Logging Interface and provides a Notification Interface to receive fault localization updates. The Trace Log Store receives trace data and forwards them to the Trace Converter. The Domain Partition Manager maintains the customizable value partitions for input messages. For instance, if a trace contains an integer input parameter $x = -173$ and the chosen domain partition for x is $\{negative, zero, positive\}$ then the Trace Converter transforms the input to $x = negative$. The transformed traces are put to the Weka Instances Store. The Decision Tree Pool utilizes the Weka J48 Classifier to maintain the set of trees. The Statistics Calculator determines quality measures for the learned classifiers, and the Training Scheduler triggers the adaptation of the tree pool to changing environments.

4. EVALUATION

In the following we evaluate different aspects of our proposed fault localization approach. We have set up a comprehensive evaluation framework as part of *Indenica*[5], a research project aiming at developing a virtual platform for service computing. The framework provides traces of large SBAs, against which we run our fault detection algorithms.

4.1 Evaluation Setup

The test traces are generated randomly, with assumed uniform distribution of the underlying random generator.

ID	$\lvert\mathbf{I}\rvert$	$\lvert\mathbf{c(i)}\rvert$, $\mathbf{i} \in \mathbf{I}$	$\lvert\mathbf{p(i)}\rvert$, $\mathbf{p(i)} \in \mathbf{P}$	$\lvert\mathbf{d}\rvert$, $\mathbf{i} \in \mathbf{I}$	$\lvert\mathbf{e}\rvert$, $\mathbf{e} \in \mathbf{E_I}$	**Fault Probability**
$S1$	5	5	10	20	$\{1\}$	$4 * 10^{-2}$
$S2$	5	5	10	20	$\{2\}$	$2 * 10^{-3}$
$S3$	5	5	10	20	$\{3\}$	$1 * 10^{-4}$
$S4$	5	5	10	20	$\{3, 3, 3\}$	$3 * 10^{-4}$
$S5$	10	10	10	100	$\{3, 4\}$	$1.001 * 10^{-6}$
$S6$	10	10	10	100	$\{4\}$	$1 * 10^{-12}$

Table 2: Fault Probabilities for Exemplary SBA Model Sizes

Table 2 shows six different SBA instances with corresponding parameter settings that are considered for evaluation. $|I|$ denotes the number of service interfaces, $|c(i)|$ is the number of concrete implementations of each interface $i \in I$, $|p(i)|$ represents the number of input parameters per interface, $|d(p)|$ is the domain size for a parameter $p \in P$, and $|E_I|$ is the number of injected incompatibilities that cause

[4] `http://www.cs.waikato.ac.nz/ml/weka/`

[5] `http://www.indenica.eu/`

the faults at runtime. The table also lists for each setting the probability that a fault occurs in a random execution.

All tests have been performed on machines with two Intel Xeon E5620 quad-core CPUs, 32 GB RAM, and running Ubuntu Linux 11.10 with kernel version 3.0.0-16.

4.2 Training Duration

First, we evaluate how many fault traces are required by the J48 classifier to pass the threshold for reliable fault detection. The scenario SBAs $S3, S2, S1$ (cf. Table 2) were used in Figure 4, 20 iterations of the test were executed, and the figure contains three boxes representing the range of minimum and maximum values. As shown in Figure 4, the number of traces required to successfully detect a faulty configuration depends mostly on the complexity (i.e., probability) of the fault with regard to the total scenario size.

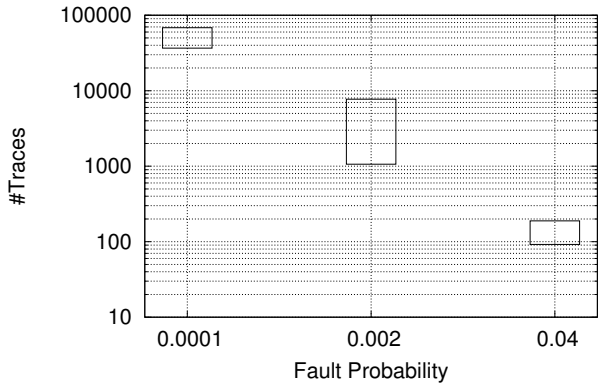

Figure 4: Number of Traces Required to Detect Faults of Different Probabilities

A single fault in the configuration $S1$ was on average detected after observing between 90 and 190 traces. If we multiply these values with the fault probability of $4 * 10^{-2}$, we get a range of 4 to 8 fault traces required for the localization. Also with more complex (unlikely) faults the relative figures do not appear to change considerably. With a fault probability of $2 * 10^{-3}$ and $1 * 10^{-4}$ the faults are detected after observing 3/16 and 4/7 minimum/maxiumum fault traces, respectively. The data suggest that there is a strong relationship between the number of required fault traces and the fault probability.

4.3 Transient Faults

As discussed in Section 2.6, our fault localization approach is designed to cope with changing environments, which is evaluated here. Figure 5 shows the performance in the presence of changing faults. The evaluation setup is as follows: Initially a fault combination $FC1$ (e.g., $\langle t_x(i_2,' premium') = false, i_3 = c_8 \rangle$) is active. At trace 33000, the implementation that causes the fault $FC1$ is repaired, but the fix introduces a new fault $FC2$ that is fixed at trace 66000. At trace 66000, another fault $FC3$ occurs, and an attempted fix at trace 88000 introduces an additional fault $FC4$, while $FC3$ remains active. At trace 121000, both $FC3$ and $FC4$ are fixed, but two new faults $FC5$ and $FC6$ are introduced to the system. The occurrence probability for each of the fault combinations ($FC1 - FC6$) is set to $2 * 10^{-3}$ (corresponding to scenario setting $S2$ in Table 2).

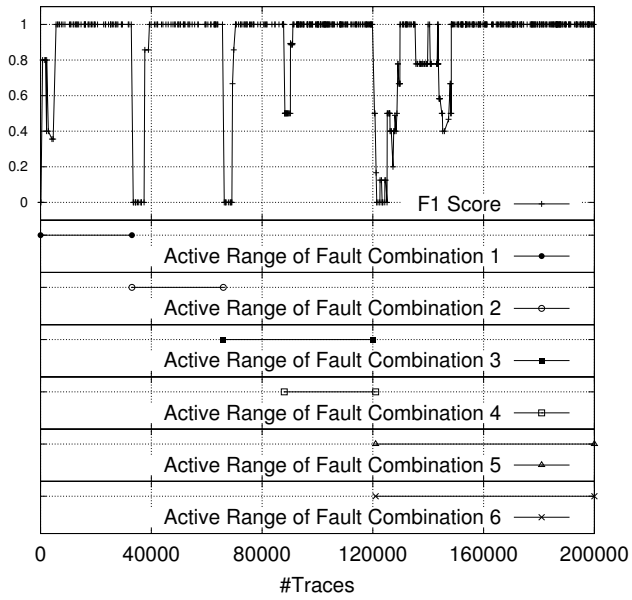

Figure 5: Fault Localization Accuracy for Dynamic Environment with Transient Faults

This scenario is designed to mimic a realistic situation, but serves mainly to highlight several aspects of our solution. After about 4000 observed execution traces the localizer provides a first guess as to the cause of the fault, but the classification is not yet correct. After around 5200 observed execution traces, the localizer was able to analyze enough error traces to provide an accurate localization result. Note that at that time, only about 6 error traces have been observed, yet the algorithm already produces a correct result. At trace 33000, the previously detected fault $FC1$ disappears and is replaced by $FC2$. Due to the pool of decision trees maintained by our localizer, $FC2$ can again be accurately localized roughly 6000 traces later. Similarly, after $FC2$ disappears, $FC3$ is localized roughly 5000 traces after its introduction.

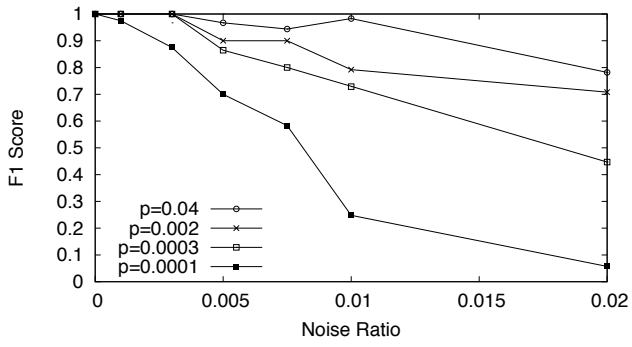

Figure 6: Noise resilience. Our approach maintains reasonable accuracy in the presence of noisy data.

The decision tree pool allows for the effective localization of new faults introduced to the system at any time. At trace 88000 in Figure 5, $FC4$ is introduced, and can again be accurately localized after observing around 5000 traces. $FC3$ and $FC4$ disappear at trace 121000 and are replaced

by simultaneously occurring errors $FC5$ and $FC6$. This situation is more challenging for our approach, as seen in the rightmost 80000 traces in Figure 5. The spikes between trace 121000 and 150000 represent different localization attempts that are later invalidated by contradicting execution traces. Finally, however, the localization stabilizes and both faults $FC5$ and $FC6$ are accurately detected.

We also evaluated the performance of our approach using different noise levels in the trace logs. Figure 6 analyzes how the F1 score develops with increasing noise ratio. The figure contains four lines, one each for the scenario settings $S1 - S4$. To ensure that the algorithm actually obtained enough traces for fault localization, we executed the localization run after 200000 observed traces.

4.4 Runtime Considerations

Due to the nature of the tackled problem, as well as the usage of C4.5 decision trees to generate rules, there are some practical limitations to the number of traces and scenario sizes that can be analyzed using our approach within a reasonable time. In the following we provide insights into the runtime performance in different configurations and discuss strategies for fine-tuning the performance.

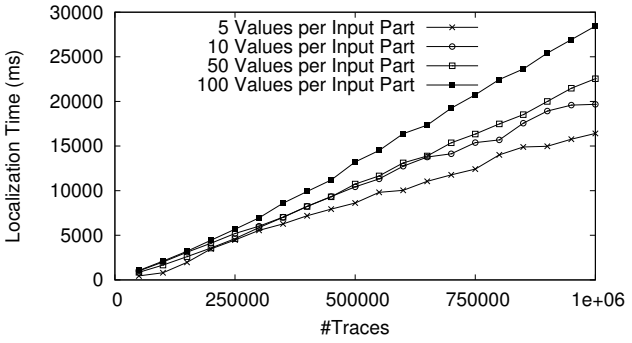

Figure 7: Localization Time for different trace window sizes in the scenario $S5$ for input sizes $|d| = \{5, 10, 50, 100\}$.

Figure 7 shows the time needed for to localize faults for various trace window sizes for the base scenario $S5$, for input sizes $|d| = \{5, 10, 50, 100\}$. The figure illustrates that the time needed for a single localization run increases roughly linearly with increasing window sizes. Larger trace windows allow the algorithm to find more complex faults. If fast localization results are needed, the window size must be kept adequately small, at the cost of the system not being able to localize faults above a certain complexity.

Furthermore, the frequency of localization runs must be considered when implementing our approach in systems with very frequent incoming traces (in the area of hundreds or thousands of traces per second). Evidently, there is a natural limit to the number of traces that can be processed per time unit. Figure 8 shows the localization speed as number of traces processed per second compared to different fault localization intervals (i.e., number of traces after which fault localization is triggered periodically) for different window sizes ($|T|$, i.e., number of considered traces).

The data in Figure 8 can be seen as a performance benchmark for the machine(s) on which the fault localization is executed. Executing this test on different machines will result in different performance footprints, which serves as a decision support for configuring window size and localization interval. For instance, if our application produces 1500 traces per second (i.e., processes 1500 requests per second), a localization interval greater than 400 should be used. Currently, the selection happens manually, but as part of our future work we investigate means to fine-tune this configuration automatically.

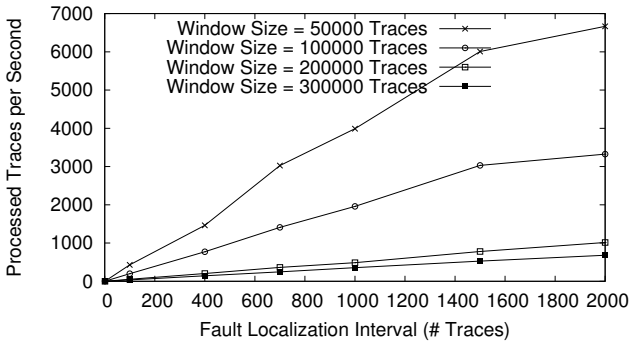

Figure 8: Localization performance in traces per second for different fault localization intervals and window sizes, using scenario $S5$

5. RELATED WORK

In this section we discuss existing approaches related to reliability, fault detection, and fault localization in SBAs and distributed systems in general.

5.1 Software Testing

Our work is related to the broad field of software testing where a plethora of approaches for fault localization have been proposed. Generally, software testing is the process of executing a program or systems with the intent of finding errors [20]. Testing approaches are often divided into white- and black-box testing. In white-box (or logic-driven) testing the internals of the software under test are visible to the tester. Black-box (input/output-driven) testing has to get along with no information about internal structure. Our problem formulation faces a black-box model in which we can observe the system behavior but have no details about the internals. Formal verification of software is an alternative to testing that is often employed in highly safety-critical environments.

Canfora et al. [4] provide an extensive overview of testing services and SBAs. The seminal work by Narayanan and McIlraith [21] was among the first to perform automated simulation and verification based on a semantic model of Web services. Another related approach has been presented in [11], which performs upfront integration testing with different combinations of concrete service implementations. Due to the huge search space, even in medium sized SBAs, their test case generation approach is not able to consider the service input and output data, whereas the efficient fault localization algorithms used in this work allow us to do so. Concluding, in software testing a system is actively executed to find problems; in this work, however, we do not control the software but we monitor its execution to localize faults and fault reasons at runtime.

5.2 Software Fault Localization

Software fault localization helps to identify bugs in software on the source code level. Oftentimes a two-phase procedure is applied: 1) finding suspicious code that may contain bugs and 2) examining the code and deciding whether it contains bugs with the goal of fixing them. Research mainly focused on the former, the identification of suspicious code parts with prioritization based on its likelihood of containing bugs [31, 17, 7]. The seminal paper by Hutchins et al. [12] introduces an evaluation environment suitable for fault localization (often referred to as the *Siemens suite*), consisting of seven base programs (in different versions) that have been seeded with faults on the source code level. Fundamental research on statistical bug isolation is presented in [17]. Decision branches are modeled as predicates, and conditional probabilities are used to compute the likelihood that a failure occurs in a certain branch.

Renieres et al. [26] present a fault localization technique for identifying suspicious lines of a program's source code. Based on the existence of a faulty run of the program and many correct runs they select the correct run that is most similar to the faulty one. Proximity is defined based on the program spectra. Then, traces of the two runs are compared and suspicious program lines are reported. This general approach is very common in software fault localization. Arguing that traditional trace proximity (literal comparison of traces) is insufficient as faults can be triggered in various ways, Liu and Han [19] introduce *R-Proximity* which regards two traces as similar if they appear to have roughly the same fault location. Guo et al. [8] propose a different similarity metric based on control flow. The metric takes into account the sequence of statement rather than just the unordered set. Our work differs from traditional software fault localization in that we do not analyze program code but only observe the runtime behavior of services. We also assume that the environment or service implementations may change during runtime, in contrast to the analysis of static code. The work in [15] assists humans in localizing software faults by visualizing test information and highlighting suspicious code statements with different color intensity. The empirical study conducted shows that single faults are evidently easier to find for humans than complex fault combinations, which strengthens the motivation for automated machine learning based fault localization, as studied here.

5.3 Monitoring and Fault Detection

Monitoring and fault detection are key challenges for implementing reliable distributed systems. Fault detectors are a general concept in distributed systems and aim at identifying faulty components. In asynchronous systems it is in fact impossible to implement a perfect fault detector [5], because faults cannot be distinguished with certainty from lost or delayed messages. Heartbeat messages can be used for probabilistic detection of faulty components; in this case a monitored component or service has the responsibility to send heartbeats to a remote entity. The fault detector presented in [27] considers the heartbeat inter-arrival times and allows for a computation of a component's faulty behavior probability based on past behavior. Steinder and Sethi [29] study fault localization in communication systems using belief networks. The approach is noise resilient and able to handle spurious events, but if fault conditions change permanently, updates in the belief network are arguably slower than using pooled decision trees. Moreover, their results indicate that fault localization time has exponential growth in the number of network nodes, whereas our centralized approach scales near-linearly in the number of traces. Lin et al. [18] describes a middleware architecture called *LIama* that advocates a service bus that can be installed on existing service-based infrastructures. It collects and monitors service execution data which enable to incorporate fault detection mechanisms using the data. Such a service bus can be used to collect the data necessary for our analysis. The major body of research in the area of monitoring and fault detection in SBAs deals with topics like SLAs (service-level agreements) [16] and service compositions rather than compatibility issues [22].

5.4 Fault Analysis and Adaptation

Fault analysis derives knowledge from faults that have been experienced. Adaptation tries to leverage this knowledge to reconfigure the system to overcome faults. Oftentimes, domain-specific knowledge is required to efficiently analyze faults and their origins (e.g., [10]). Zhou et al. [32] have proposed GAUL, a problem analysis technique for unstructured system logs. Their approach is based on enterprise storage systems, whereas we focus on dynamic service-based applications. GAUL uses a fuzzy match algorithm based on string similarity metrics to associate problem occurrences with log output lines. The aim of GAUL differs from our approach since we assume the existence of structured log files and focus on the localization of faulty configuration parameters. Control of SOAs mostly relies on static approaches, such as predefined policies [23]. Techniques from artificial intelligence can be used to improve management policies for SBAs during runtime. For instance, Markov decision processes represent a possible way for modeling the decision-making problems that arise in controlling SBAs. Markov decision processes and algorithms to solve them have been shown effective in reducing the impact of defects in service implementations by adapting the SBA at runtime [14, 13]. In this work we focus on fault localization rather than on how to react in the face of faults.

6. CONCLUSION

In this paper we describe a fault localization technique that is able to identify which combinations of service bindings and input data cause problems in SBAs. The analysis is based on log traces, which accumulate during runtime of the SBA. A decision tree learning algorithm is employed to construct a tree from which we extract rules, describing which configurations are likely to lead to faults. For providing a fine-grained analysis we do not only consider the service bindings but also data on message level. This allows to find incompatibilities that go beyond "service A has incompatibility issues with service B" leading to rules of the form "service A has incompatibility issues with service B for messages of type C". Such rules can help to safely use partial functionality of services. We present extensions to our basic approach that help to cope with dynamic environments and changing fault patterns. We have conducted experiments based on scenario traces of realistic size. The results provide evidence that the employed approach leads to successful fault localization for dynamically changing conditions, and is able to cope with the large amounts of data that accumulate by considering fine-grained data on message level.

As future work we plan to extend our approach beyond the pure fault localization aspects; in particular, we will use the extracted rules for guiding automated reconfiguration when a fault occurs. Furthermore, we intend to integrate test coverage mechanisms that help to actively investigate faults. This can be used for systematic test execution of insightful configurations and input requests which further narrow down the search space of possible fault reasons.

Acknowledgment

This work is partially supported by the Austrian Science Fund (FWF): P23313-N23, and has has received funding from the European Commission's Seventh Framework Programme (FP7) under grant agreement 257483 (Indenica).

7. REFERENCES

[1] D. W. Aha. Tolerating noisy, irrelevant and novel attributes in instance-based learning algorithms. *Int. Journal of Man-Machine Studies*, 36(2):267–287, 1992.

[2] R. Baeza-Yates and R.-N. Berthier. *Modern information retrieval*. Addison-Wesley, 1999.

[3] B. Blau, J. Kramer, T. Conte, and C. van Dinther. Service value networks. In *IEEE Conference on Commerce and Enterprise Computing*, 2009.

[4] G. Canfora and M. Di Penta. Testing services and service-centric systems: challenges and opportunities. *IT Professional*, 8(2):10–17, 2006.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[6] M. R. Chmielewski and J. W. Grzymala-Busse. Global discretization of continuous attributes as preprocessing for machine learning. *International Journal of Approximate Reasoning*, 15(4):319 – 331, 1996.

[7] H. Cleve and A. Zeller. Locating causes of program failures. In *27th International Conference on Software Engineering (ICSE)*, pages 342–351. ACM, 2005.

[8] L. Guo, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In *Int. Conference on Compiler Construction*, 2006.

[9] G. Hripcsak and A. S. Rothschild. Agreement, the f-measure, and reliability in information retrieval. *Journal of the American Medical Informatics Association*, 12(3):296–298, 2005.

[10] W. Hummer, C. Inzinger, P. Leitner, B. Satzger, and S. Dustdar. Deriving a unified fault taxonomy for event-based systems. In *Int. Conference on Distributed Event-Based Systems (DEBS)*, pages 167–178, 2012.

[11] W. Hummer, O. Raz, O. Shehory, P. Leitner, and S. Dustdar. Test coverage of data-centric dynamic compositions in service-based systems. In *Int. Conference on Software Testing, Verification and Validation (ICST)*, pages 40–49, 2011.

[12] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *16th Int. Conference on Software Engineering (ICSE)*, 1994.

[13] C. Inzinger, B. Satzger, W. Hummer, and S. Dustdar. Specification and deployment of distributed monitoring and adaptation infrastructures. In *Workshop on Performance Assessment and Auditing in Service Computing at ICSOC'12*, 2012.

[14] C. Inzinger, B. Satzger, W. Hummer, P. Leitner, and S. Dustdar. Non-intrusive policy optimization for dependable and adaptive service-oriented systems. In *ACM Symposium on Applied Computing*, 2012.

[15] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *24th International Conference on Software Engineering (ICSE)*, pages 467–477, 2002.

[16] P. Leitner, W. Hummer, and S. Dustdar. Cost-based optimization of service compositions. *IEEE Transactions on Services Computing*, PP(99):1, 2011.

[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 15–26. ACM, 2005.

[18] K.-J. Lin, M. Panahi, Y. Zhang, J. Zhang, and S.-H. Chang. Building accountability middleware to support dependable SOA. *Internet Computing*, 13(2), 2009.

[19] C. Liu and J. Han. Failure proximity: a fault localization-based approach. In *Int. Symposium on Foundations of Software Engineering (FSE)*, 2006.

[20] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing, Third Edition*. Wiley, 2011.

[21] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Int. Conference on World Wide Web*, 2002.

[22] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11), 2007.

[23] T. Phan, J. Han, J.-G. Schneider, T. Ebringer, and T. Rogers. A survey of policy-based management approaches for service oriented systems. In *19th Australian Conference on Software Engineering*, 2008.

[24] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[25] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., 1993.

[26] M. Renieres and S. Reiss. Fault localization with nearest neighbor queries. In *18th Int. Conference on Automated Software Engineering*, pages 30–39, 2003.

[27] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer. A new adaptive accrual failure detector for dependable distributed systems. In *ACM Symposium on Applied Computing*, pages 551–555. ACM, 2007.

[28] T. Segaran. *Programming Collective Intelligence*. O'Reilly Media, 2007.

[29] M. Steinder and A. S. Sethi. Probabilistic fault localization in communication systems using belief networks. *Transactions on Networking*, 12(5), 2004.

[30] E. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, 1991.

[31] W. E. Wong and V. Debroy. Software fault localization. *Part of the IEEE Reliability Society 2009 Annual Technology Report.*, 2009.

[32] P. Zhou, B. Gill, W. Belluomini, and A. Wildani. Gaul: Gestalt analysis of unstructured logs for diagnosing recurring problems in large enterprise storage systems. In *29th IEEE Symposium on Reliable Distributed Systems*, pages 148 –159, 2010.