Towards Identifying Root Causes of Faults in Service-Based Applications

Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner and Schahram Dustdar
Distributed Systems Group, Vienna University of Technology, Argentinierstrasse 8/184-1, A-1040 Vienna, Austria
{lastname}@dsg.tuwien.ac.at, http://dsg.tuwien.ac.at/

***Abstract*—**
In this paper we study fault localization techniques for identification of incompatible configurations and implementations in service-based applications. We propose an approach using pooled decision trees for localization of faulty service parameter and binding configurations, explicitly addressing temporary and changing fault conditions.

I. 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. In many industries, the technical interfaces of these services are nowadays governed by industry standards, specified by bodies such as the TM Forum[1]. Hence, integration of services provided by different business 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., dynamically select the most suitable implementation of a given standardized interface based on fluid business 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 the abstract, 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. Our approach takes into account not only the actual service implementations themselves, but also the received input and the produced output data of implementations.

[1]http://www.tmforum.org/browse.aspx

II. Fault Localization Approach

An SBA uses a set of industry standard service interfaces. For each interface, there is usually a set of implementations available. Each interface defines the domains of possible input parameters. Our fault localization technique is based on execution traces containing the binding context of all used interfaces, observed input parameter values, as well as an indicator signifying the *success* or *fault* of the request.

To estimate the number of possible traces for a medium sized application, consider an imaginary SBA using 20 interfaces, 3 candidate implementations per interface, 4 input parameters per service, and 10 possible data values per parameter. The theoretical number of possible executions in this SBA is $3^{20} * (10^4)^{20} = 3.48678 * 10^{89}$. 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 to achieve this:

1) The first manual preprocessing step is to decide, based on domain knowledge about the SBA, which attributes are relevant for fault localization. For instance, unique attributes such as $requestID$ should 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 program faults often depend on a range of values with common characteristics [1]. Partition testing strategies hence divide the domain of values into multiple sub-domains and treat all values within a sub-domain as equal. 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 automate this procedure (e.g., [2]).

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 [3], a popular technique in machine learning. Note that decision trees are usually used for classification, which means to learn rules from a set of training instances with the aim of predicting the class attribute of a new instance. However, our purpose is not classification because in our problem formulation the value of the class attribute ($success$ or $fault$) is known for each trace instance; instead, we are interested in learning a decision tree to obtain the rules which apply to a particular value of the class attribute ($fault$). The decision tree with binary split [3] is used to automatically derive incompatible attribute values. Basically, the procedure is to loop over all $fault$ leaf nodes and to create a combination of attribute assignments along the path from the leaf to the root node.

In real-life systems which are influenced by various external factors, we have to deal with temporary and changing faults. We make use of decision tree algorithms to cope with such temporary faults, which can be considered as noise in the training data (e.g., [4]). In cases where the root causes of failures within an SBA change from time to time, we need a mechanism to let the machine learning algorithms forget old traces and train new decision trees based on fresh data. To this end, we assess the accuracy of an existing classification model using the well-known measures *precision*, *recall*, and *F1 measure*, which is defined as the harmonic mean of precision and recall.

Figure 1 illustrates a representative sequence of execution traces ($\{t_1, t_2, t_3, \ldots\}$). The top of the figure shows the trace results ($r(t_x)$), where "S" represents $success$ and "F" represents $fault$. As the traces arrive with progressing time we utilize the C4.5 algorithm [3] to infer decision trees from the data. At time point 1, the decision tree d_1 is initialized. 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 d_i is assessed by classifying new incoming traces x (denoted as $rc(d_i, x)$). In Figure 1 correct classifications are printed in normal text, while incorrect classifications are printed in bold underlined font.

Four particularly interesting time points (a, b, c, d) are marked in Figure 1. In time point a the tree d_1 misclassifies trace t_a as a false positive. This misclassification triggers the parallel training of a new decision tree d_2 based on the traces starting with t_a. A false negative by d_2 happens in time point 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. Time point c contains another false positive by 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. At time d the changing 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 stored for reference, but are not trained with further data to save computing power.

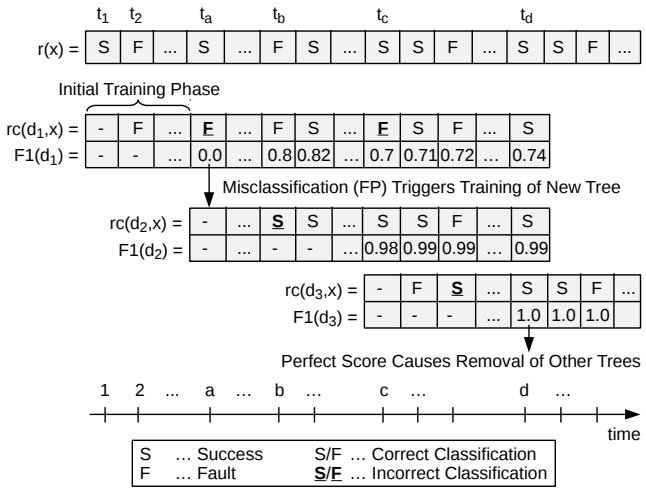

Fig. 1. Maintaining Multiple Trees to Cope with Changing Faults

III. Conclusion

In this paper we describe a fault localization technique that is able to identify problematic combinations of service bindings and input data in SBAs. The analysis is based on log traces, which accumulate during runtime of the SBA. A decision tree learning algorithm is used to create a pool of trees from which we extract rules, identifying configurations and inputs that likely lead to faults.

As future work we plan to extend our approach beyond the pure fault localization aspects; in particular, we will further validate our technique with extensive evaluations and aim to integrate the extracted rules for guiding automated reconfiguration using reinforcement learning [5] when faults occur. Furthermore, we intend to integrate test coverage mechanisms (e.g., [6]) 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 was partially funded by the European Commission's FP7 project 257483 (Indenica).

References

[1] E. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 703–711, 1991.

[2] M. R. Chmielewski and J. W. Grzymala-Busse, "Global discretization of continuous attributes as preprocessing for machine learning," *Intl. Journal of Approximate Reasoning*, vol. 15, no. 4, pp. 319 – 331, 1996.

[3] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993.

[4] D. W. Aha, "Tolerating noisy, irrelevant and novel attributes in instance-based learning algorithms," *Int. J. Man-Mach. Stud.*, vol. 36, no. 2, pp. 267–287, 1992.

[5] 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, pp. 504–510.

[6] W. Hummer, O. Raz, O. Shehory, P. Leitner, and S. Dustdar, "Test coverage of data-centric dynamic compositions in service-based systems," in *Intl. Conference on Software Testing, Verification and Validation*, 2011.