Delft University of Technology, Bachelor Seminar of Computer Science and Engineering

Inferring DFAs from Log Traces Using Community Detection

Tommaso Brandirali
Delft University of Technology
Delft, The Netherlands

Mitchell Olsthoorn
Delft University of Technology
Delft, The Netherlands

Annibale Panichella
Delft University of Technology
Delft, The Netherlands

ABSTRACT

Large software systems today require increasingly complex models of their execution to aid the analysis of their behavior. Such execution models are impractical to compile by hand, and current approaches to their automated generation are either not generalizable or not scalable enough. This paper addresses this problem with a new approach based on the interpretation of log traces. We analyze the effectiveness of using community detection algorithms for generating system execution models from structured datasets of log samples. This approach first models sets of log traces as tree-shaped automata, and then uses graph clustering algorithms to reduce such tree representations down to more concise models. This research focuses on analysing the quality of the generated models in terms of conciseness, accuracy, recall and scalability. Testing was performed on data samples from the XRP network, a blockchain-based payment system. During implementation of a proof of concept, multiple challenges arose which limited the ability of our study to fully evaluate the approach's effectiveness. The partial results obtained show poor performance, both in terms of runtime and in accuracy of generated models. Due to the limitations of the evaluation performed, the results are to be considered exploratory and require further testing.

1 INTRODUCTION

As enterprise software systems become increasingly complex and multi-layered, analysis of their high-level behavior is becoming increasingly challenging. While static analysis and testing are crucial to detecting low level bugs and flaws before deployment, they can struggle to detect deviations from intended behavior in long execution paths. Such high-level dynamic execution analysis requires system models to be built, either manually or programmatically, to compare empirical execution data against in order to spot unintended behavior. Building such models is becoming increasingly difficult as the complexity of software systems increases, and current approaches to their generation aren't sufficiently generalizable.

Popular techniques for dynamic execution analysis today include profiling, which builds system models by periodically sampling the state of the program, and tracing, which makes use of code instrumentation tools to record a detailed history of the program's call stack. These techniques provide structured data about a system's execution with various degrees of accuracy. However, both these techniques perform poorly in high-performance environments: both of them introduce overhead to the execution which impacts the overall performance, and the latter also produces large amounts of data at high rates whose processing can add undue strain on the underlying machine. Additionally, both these methods produce data which, while being structured, is often too low-level to provide useful insights for the system's maintainers [16]. Another option for collecting data about a system's execution is log trace analysis. This approach does not have the same performance issues as the previously mentioned approaches, however it introduces new challenges for data interpretation. Log data is less structured and generalisable than execution profiles or traces, due to the variety of logging schemes and frameworks used in practice. All the formerly mentioned approaches still require the compilation of high-level software models as comparison tools for the purpose of detecting unwanted behavior. Our research focuses on tackling the generation of such models for the analysis of log data, due to the higher scalability of log data collection.

Log data interpretation is an open problem in the field, with promising research being conducted on various techniques. The goal of these techniques is to process low-level log samples to generate high-level system models, usually in the form of Finite Automata. These approaches mostly use combinations of exact and heuristic strategies for system model synthesis. One such approach has been developed by Verwer et al. to build concise state models from log trace samples using a greedy state-merging algorithm [19], in a tool called *flexfringe*. Roelvink et al. have further tested the latter approach by comparing empirical models learned by *flexfringe* with theoretical models [14]. Another approach has been proposed, also by Verwer et al., using a combination of greedy heuristics and satisfiability solvers [9]. Both these methods obtain promising results by reducing the system modeling problem to a problem of synthesizing Deterministic Finite Automata (DFAs). This synthesis process takes as input a structured representation of the sample data as branches of a tree-shaped automaton (known as a Prefix Tree Acceptor), and reduces this prefix tree down to a concise DFA by iteratively merging closely-related states. Despite the progress made by such approaches in tackling the problem, all of them have worse-than-linear time complexities and therefore scale poorly with the amount of logs. Further research is needed to improve the performance and scalability of such algorithms.

This research attempts to tackle the DFA synthesis problem with a new approach, namely: using graph clustering algorithms on the prefix trees, with a state distance function based on the empirical probability of state transitions. This approach assumes that the topology of a prefix tree automaton built from samples of log traces encodes meaningful data about the high-level execution behavior of the program that generated the samples. This research aims to test this assumption by evaluating the accuracy of the models produced by this approach, and the performance of the associated algorithm. Specifically, evaluation is conducted in terms of the size reduction of the models produced (compression), the accuracy of the models in correctly identifying valid and invalid log traces (recall and specificity), and the runtime complexity of the algorithm. A proof-of-concept implementation for this approach was developed and tested on datasets of log traces from the `rippled` server, the core software infrastructure of the XRP network, a blockchain-based payment platform.

Our study obtained poor results for all three evaluation criteria. Compression was found to be minimal, accuracy and recall were found to be comparable to the non-compressed prefix trees due to the model not recognizing any unseen data, and the runtime for the classification of unseen data was found to vary by multiple orders of magnitude depending on the models' unique topology. These unsatisfactory results were partly due to technical challenges encountered during the implementation, which could not fully be overcome within the timeframe of this study. Due to these issues encountered, our results validity cannot be fully ensured, and they are to be considered exploratory results until further testing is done on this approach. We consider the main contribution of this paper to be the analysis of the challenges involved in implementing this approach, which could be useful in further research on similar problems.

Section 2 of this paper describes the practical setting of our implementation and tests, as well as the theoretical background used in the development of the algorithm. Section 3 describes in detail our approach and the development of the tool we used to perform the tests and evaluation. Section 4 introduces the research questions and details the setup of the empirical tests. Section 5 describes the results obtained from the evaluation and attempts to explain their nature. Section 6 discusses the challenges encountered during the implementation and their effect on the results. Section 7 discusses possible threats to the validity and generalisability of our results, while Section 8 describes the measures taken to ensure the reproducibility of our results. Finally Section 9 discusses the contributions of this paper and suggests possible questions for future research on the topic.

2 BACKGROUND AND RELATED WORK

2.1 Execution Monitoring

Program Execution Monitoring is a well known problem and research field in computer science, which deals with extracting information from potential data about the sequence of states that a program traverses during its execution [12]. It is often referred to as *dynamic analysis*, as opposed to *static analysis* which mainly deals with the interpretation of source code. Dynamic analysis techniques generally involve two main phases: gathering data from the software under scrutiny, and then interpreting such data with the goal of achieving some context-specific definition of "program comprehension" [5]. It should be noted that dynamic analysis has an inherent limitation in its necessary incompleteness: it relies on the analysis of small sample sets from large, possibly infinite execution domains [5]. Dynamic analysis approaches are therefore heuristic approaches, rather than exact ones. In contrast with static analysis approaches, however, dynamic ones are able to find subtler and more high-level flaws in programs than those which static analysis can identify. Additionally, some dynamic analysis techniques do not require access to the program source code, and can therefore be applied to closed source modules and dependencies.

2.2 DFA Inference

The problem at the core of this research is a version of the model inference problem first proposed by Moore in 1956 [10]. It concerns the synthesis of a Deterministic Finite-state Machine (DFA) from a set of empirical labeled data sampled from a regular language, such that the DFA at least recognizes the sampled subset of the language. This problem is trivial when there are no constraints on the nature of the final automaton: indeed, it is sufficient to build the prefix tree acceptor for the samples to obtain some DFA that meets the aforementioned criteria. However, in practice, there usually are constraints on the final automata, such as having a fixed number of states. This latter version of the problem has been shown to be NP-complete [8]. Additionally, the related problem of finding an accepting DFA with the minimum number of states has been proven impossible to approximate with any polynomial [11], therefore recent research on the topic has focused on heuristic approaches rather than exact ones.

A related goal often found in applications of the DFA Inference problem is that the generated automata should generalize to unseen data, provided it is generated by the same stochastic process that generated the training data. This is indeed the case with our log interpretation problem: we aim to produce automata that accurately model the execution of the software which produced the log samples, which in practice means that our models should be able to accurately classify unseen log traces from the same program. To achieve this, two assumptions have to be made, namely: that log statements do contain meaningful information about the state of the underlying program, and that the selection of execution paths from the execution domain approximates a stochastic process.

Many algorithms for DFA Inference follow the general paradigm of the ALERGIA algorithm developed in 1994 [4]. The purpose of the algorithm is to generate a non-trivial DFA accepting a regular language, based on a complete presentation of the language in the form of stochastically selected samples. It starts by building the trivial accepting DFA, the prefix tree acceptor of the sample set. A prefix tree acceptor is a tree-shaped automaton recognizing a finite language of words, built from the empirical samples of the language rather than a general description of it. Whenever two words in the language share a prefix of one or more tokens, their accepting paths in the prefix tree overlap in as many nodes as tokens in the shared prefix. The ALERGIA algorithm builds the prefix tree and then enriches it with the relative empirical probabilities of the transitions coming out of each node. It then iteratively merges nodes in the tree with a greedy strategy: it uses the transition probabilities to calculate a confidence score for the equivalence of two nodes and merges pairs of nodes in order of decreasing confidence until some stopping criterion. While this algorithm is not often used in practice, due to the requirement that the sample set be a complete presentation of the language, it provides a popular algorithmic strategy for problems related to DFA inference.

2.3 Community Detection

Community detection is the process of finding clusters of closely related nodes in a graph [7]. Community detection algorithms generally use some definition of distance between nodes in order to find clusters that minimize the distance between any two nodes in the same cluster and maximize the distance between nodes in different clusters. Hierarchical clustering is a class of community detection algorithms that build a bottom-up hierarchy of nested clusters. The product of these algorithms is commonly known as a *dendrogram*, and it is usually modeled by a list of merges between two nodes, in order of increasing merge cost. The dendrogram defines n merges where n is the number of nodes in the graph. By truncating the dendrogram at n - k merges we obtain k - 1 clusters.

Finite automata can be represented as directed graphs, where nodes represent states and edges represent transitions between states. This representation allows us to use graph-based algorithms, including community detection, to extrapolate information from a DFA as we would from any directed graph. The computation of a finite automaton on an input word in this case is equivalent to finding a path through the graph where each node matches a subsequent token from an input sample. In the rest of this paper, we will often use the names nodes and states, edges and transitions, words and log traces, and graphs and automata interchangeably, since they represent the same entities within the scope of this problem. Community detection in this context can then be seen as an algorithm for finding fitting superstates in the DFA. However, as mentioned, clustering algorithms require some notion of distance between nodes in order to evaluate pairwise clustering of nodes. In order to use clustering algorithms on DFAs, these have to be enriched with some transition-level data that can quantify the degree of relatedness of neighbouring nodes. The ALERGIA algorithm mentioned in section 2.2 provides one such metric: relative transition probabilities. We can define the distance between a state and one of its children as the inverse of their transition probability. This implies that states with a high probability of occurring in succession in the sample data would be considered closer to each other and more likely to be placed in the same superstate. Conversely, states with a low probability of occurring in succession are considered farther from each other and more likely to be placed in different superstates.

3 APPROACH

This section details our approach to the problem at hand, the intuitions underlying the logical steps and choices made, as well as our approach's connections to related problems.

3.1 Syntax Model

Our approach to the interpretation of log traces relies on the identification of log statements originating from the same location in the source code as representing the same program state. In other words, it is necessary for such an approach to incorporate a log syntax model capable of identifying equivalent log statements, defined as originating from the same line in the source code. This problem would be trivial if log statements were exclusively composed of static text. However, in practice, log statements can contain dynamic text representing program variables, which carries little to no information about the program state. The syntax model must therefore address the problem of identifying the static components of log statements, in order to then classify different log statements as equivalent based on having the same static components. If the source code of the program being modeled is not available for review, the assumption can be made that log statements with the same static components found in the same position in an execution path represent the same low-level state of execution of the program. Research is being conducted on automating this process using various methods, with promising approaches including counting the frequency of n-grams [6], and using Recurring Neural Networks to identify semantically different parts of log statements [13]. For modules with limited amounts of unique log statements, this model could also be built manually. In any case, such a syntax model is needed to structure the sample data into a format which encapsulates useful information for the modeling of program states.

3.2 Prefix Tree

While the syntax model allows for structuring the sample data at the level of single log statements, modeling the entire execution behavior of the program requires building a structuring of the data which encapsulates the relationship between unique log statements within full execution paths. A popular strategy used in existing approaches to the DFA inference problem is the one introduced by the ALERGIA algorithm [4], which involves building a prefix tree acceptor of the available traces. A prefix tree acceptor can be built by creating a placeholder root and then iteratively scanning each word in the sample set token by token, searching for a path in the tree matching the current prefix of tokens. Whenever no path is found for the current prefix, a new child is added to the last matched node. Whenever a word is completely scanned, the last node matched or created is marked as a leaf. When the prefix tree is fully built, it will have exactly one matching path for each word in the language. By definition, the prefix tree is entirely specific to the language samples it is built from: it accepts all and only the words that were used to create it. Algorithm 1 shows the pseudocode for a prefix tree generation algorithm.

While the prefix tree can effectively represent the sample set in a single, searchable data structure, it can contain a significant amount of redundancy. This is because the words in its language can contain subsequences made from a single repeated token. This is especially true in the case of log traces, as loop statements in source code can result in the same log statement being printed multiple times. This redundancy can be mitigated by substituting sequences of equivalent states with a single such state with a self-looping transition. Notably, this implies some loss of information about loop iteration constraints, and their influence on legal paths through the node. In our approach, we assumed that such an information loss would be negligible in the context of large traces. The gains resulting from such an optimization are entirely dependent on the amount of redundancy in the training samples, so its use should be evaluated on a case-by-case basis.

3.3 Clustering

While the prefix tree does model the input dataset, it is neither a concise nor generalizable model, therefore it must be processed with the goal of reducing it to a more concise and more generalizable representation of the input dataset. This process is where most approaches to the DFA inference problem differ between each other. Most such algorithms iteratively merge states in the DFA based

Algorithm 1: Algorithm to generate a prefix tree from a set of traces

```
Input: A SyntaxTree instance $ST$ and a list of log traces $L$,
       where each trace $t$ is a list of strings
Output: A PrefixTree instance
init PrefixTree $PT$ with placeholder root $r$
foreach $trace \in L$ do
    CurrentState ⟵ $r$
    foreach $log \in trace$ do
        CurrentChildren ⟵ CurrentState.children
        LogState ⟵ node ∈ CurrentChildren | node == log
        if $LogState$ is null then
            add $log$ as new child of CurrentState
            CurrentState ⟵ $log$
        else
            if $LogState == CurrentState$ then
                add edge from CurrentState to CurrentState
            else
                CurrentState ⟵ LogState
            end
        end
    end
    mark CurrentState as leaf
end
```

on some fitness criterion and within certain boundaries for which merges are considered legal. This fitness criterion is what defines the unique intuition of each algorithm, while the criteria for merge legality define the requirements of the merging algorithm. Merging states means replacing two states with a single state containing a union of the data sets contained in each of the original states, and having transitions from each parent and to each children of both the original nodes.

The intuition behind our approach is to use the relative, empirically estimated, transition probabilities as a measure of distance between nodes. These probabilities are measured using the number of passes through each transition during the construction of the prefix tree. For each transition, the number of passes is divided by the sum of passes through the transition's origin node, to obtain the (empirically estimated) conditional probability of choosing one child given the current node. In our approach, however, we want to consider nodes with a high probability of occurring in succession as closer together, therefore, we decided to use the complements of probabilities as a distance measure. This leads us to the following definition of node distance:

$$distance(A, B) = 1 - \left(\frac{passes_{AB}}{\sum\limits_{X \in outgoing_A} passes_{AX}} \right) \tag{1}$$

Where A and B are the origin and destination node of the transition, $passes_{AB}$ is the number of passes on a transition from A to B, and $outgoing_A$ is the set of child nodes of A.

This distance definition allows us to use topological clustering algorithms for graphs to guide the selection of merges in the reduction of the prefix tree. What we need from such algorithms is a definition of a hierarchy of merges to be performed in succession and in order of decreasing fitness, so that the merging may be stopped once it crosses a threshold of desirability. Various clustering algorithms are available which fulfill exactly such a need, namely: hierarchical clustering algorithms. Such algorithms produce data structures called *dendrograms*, represented as lists of merges ordered by decreasing fitness. These dendrograms can be used to guide the iterative merging of the prefix tree. We opted to use the Louvain method [1] as a clustering algorithm due to its performance and wide availability of implementations.

Clustering algorithms generally do not account for the requirements of DFAs, and applying their merges on DFAs can result in the introduction of non-deterministic transitions (nodes having multiple equivalent children). This requires the use of a determinization subroutine after each merge, which would resolve any non-determinism in the graph by merging equivalent children together. If this is applied, a single merge from the dendrogram can result in more than two nodes being merged together, as the determinization can affect multiple pairs of nodes. As a result, not all merges from a clustering dendrogram will be applicable, as some will have been invalidated by previous determinizations. Algorithm 2 describes the pseudocode for this prefix tree reduction. It assumes that a dendrogram is given as a list of tuples, each tuple representing the nodes to be merged. The stopping condition for the algorithm is to be chosen on a case-by-case basis.

Algorithm 2: Algorithm to reduce a prefix tree using a dendrogram

```
Input: A PrefixTree instance $PT$ and a dendrogram $D$
Output: The reduced graph
foreach $s1, s2 \in D$ do
    if $s1, s2$ not yet merged then
        MERGE($s1$, $s2$)
        DETERMINIZE($s1$, $s2$)
    end
    if stopping condition then
        break
    end
end
```

4 EMPIRICAL STUDY AND TESTING

This section will detail our research questions, together with the experimental context and benchmarks used to define those questions. This section will also detail steps taken during the implementation in order to allow for benchmarking to take place, and which parameters have been used during benchmarking.

4.1 Research Questions

We can now define with more precision the questions our research aims to answer:

RQ1 *How effective is our approach at reducing prefix trees, in terms of the compression ratio of the generated models?*
RQ2 *How accurate are the models produced by our approach, in terms of specificity and recall evaluated on unseen test data?*
RQ3 *How efficiently can our approach's generated models be used to classify unseen traces, in terms of runtime?*

4.2 Evaluation Criteria

The goal of this research is to evaluate the quality of models generated using the algorithm described in the previous section, as well as the performance of the algorithm itself. The research questions will be evaluated in terms of the following factors: the size of the generated models in comparison to the size of the original dataset (Q1), the accuracy of the models in classifying valid and invalid unseen traces (Q2), and the runtime of the program when using the models to classify unseen data (Q3). The following subsections will detail the definition and benchmarking of each of these criteria:

4.2.1 Size. One of the core goals in most approaches to this problem is to generate concise models: models with a significantly smaller size than the sample dataset. For our first research question, rather than measuring an absolute size, what we are interested in is the reduction in size of the models compared to a baseline reference, in other words: the compression ratio. As this reference, we chose to use the size of the initial Prefix Tree, measured in number of nodes. This results in the following formula for the compression, where PT is the initial Prefix Tree, M is the reduced model, and $size()$ is a function that returns the number of nodes in the graph:

$$compression = \frac{size(PT)}{size(M)} \tag{2}$$

4.2.2 Specificity and Recall. For the second research question we aim to evaluate the accuracy of our models in classifying unseen data. In order to achieve this we need to define two classes of unseen data: positive traces and negative traces. Positive traces are valid unseen traces produced by the same system, which are expected to be positively recognized by a model of the system. Negative traces are invalid traces not directly generated by the source system, which we expect the model to not recognize. While positive traces can simply be gathered by splitting the original dataset into a training and a test set, negative traces have to be procedurally generated for this purpose. In order to do this, we used an approach similar to the one used by Roelvink [14]: we took a subset of traces from the training set, and to each one applied a randomized number of mutations from 1 to 3. Each mutation is an operation on one or more sections of lines in a trace, where a section is defined as a set of subsequent lines having the same template. The type of each mutation was also randomly selected between three ones: deletion, which deletes a section outright; swap, which swaps two adjacent sections; random swap, which swaps two random sections in the trace. We additionally tested these invalid traces using a computed prefix tree in order to ensure that they were indeed invalid and did not represent alternative valid execution paths.

These two test datasets are the basis for our evaluation of accuracy. We measure this by using the generated models to classify each test trace as either a true or false trace, and then comparing these results to the original classes. This process yields a final classification of the test results into one of four categories: True Positives (TP), False Positives (FP), True Negatives (TN), False Negatives (FN). Table 1 shows the definition of these result classes.

Table 1: Overview of the result classes

	Result: True	**Result: False**
Positive	Valid traces classified as valid (TP)	Invalid traces classified as valid (FP)
Negative	Invalid traces classified as invalid (TN)	Valid traces classified as invalid (FN)

This latter classification allows us to compute two measures which define the accuracy of our models: specificity and recall. Specificity is defined as:

$$specificity = \frac{|TN|}{|TN| + |FP|} \tag{3}$$

and it represents the fraction of negative traces which the model correctly classifies as negatives. Recall is defined as:

$$recall = \frac{|TP|}{|TP| + |FN|} \tag{4}$$

and it represents the fraction of positive traces which the model correctly classifies as positives.

4.2.3 Runtime. For the third research question we aim to evaluate the speed of our implementation when using the models for their classification purposes. This criterion has to be evaluated in relation to the size of the input dataset used for the benchmarking. This is a crucial criterion since it significantly impacts the usability of this approach in a production environment. Rather than evaluating the runtime measurements in absolute terms, we chose to plot their growth against another evaluation parameter such as the size of the training set and the compression ratio of the model.

4.3 Evaluation Benchmark

Our approach was evaluated on a dataset of log traces from the XRP Ledger. The XRP Ledger is a decentralized blockchain-based payment system based on the Ripple Consensus Protocol [15]. The core software is the `rippled` server, which acts as a node in the network and can serve multiple purposes, including: submitting transactions, validating transactions, and managing API access to the ledger history. The dataset used for the proof-of-concept implementation and benchmarking of our approach consisted of log traces from a `rippled` server instance and is available at [3]. The `rippled` server is open source, and is distributed under a mix of copyleft licenses. Due to its critical role in maintaining trust within the network, it is crucial that the execution behavior of the server be stable and predictable. Unexpected or unusual behavior threatens the trust intrinsic to the network and therefore must be promptly identified and addressed. For this purpose, the server can produce large amounts of semi-structured logs at different severity levels. However, this raises a different problem: manual interpretation of these logs is unfeasible due to their scale. While automatically identifying errors or warnings is trivial, the same cannot be said for unusual and unintended execution paths which do not trigger internal consistency checks. This latter case requires building a model of the software's execution domain which is capable of detecting unusual paths, and which can be used for Program Execution Monitoring. For these reasons the `rippled` server constituted an ideal subject for testing our approach. The `rippled` server consists of various different modules, but for the scope of this research we exclusively focused on the Consensus Protocol module, as its execution follows a well-defined state machine with single entry and termination states.

4.4 Implementation

The implementation was done in Python 3.8 and is available at [17]. The implementation uses the `scikit- network` [2] library API to generate the clustering dendrogram from the Prefix Tree. During the implementation, we encountered technical and practical challenges related to the determinization of nodes after merging, which could not be fully overcome within the available time. These issues will be discussed in depth in section 6. Due to them, our generated models were not completely determinized and did contain non-deterministic transitions, which had a significant impact on the runtime complexity of the evaluation. As a result, our evaluation was limited to small sample sizes.

4.5 Evaluation Setup

All of the empirical testing was conducted on a Windows machine with an Intel i7 12-core processor, with a base speed of 2.21 GHz.

Due to the runtime issues described in the section 6, the evaluation of runtime, specificity, and recall had to be limited to relatively small trees and a limited number of merges. Our evaluation procedure consisted of *n* iterations of the following steps:

(1) Select a new set of *m* traces as training set
(2) Build the prefix tree from the training set
(3) Build the clustering dendrogram from the prefix tree
(4) Iteratively perform the first *j* merges from the dendrogram, every *k* merges perform the following:
    (a) Evaluate specificity and recall of the tree
    (b) Estimate runtime of the last evaluation
(5) Calculate final size of the model

The evaluation performed in step 4a consists of classifying 100 positive and 100 negative traces with the model at its current state, then calculating the specificity and recall values as described in section 4.2.2. The test trace sets were the same for each iteration, and were disjoint from any of the training sets. Results are discussed in section 5.

The aforementioned evaluation parameters *n*, *m*, *j*, and *k* were chosen in order to maximize the scope of the evaluation within the limitations of the evaluation time's exponential growth. *m* and *j* in particular were the main parameters affecting this growth, and therefore had to be manually tuned accordingly. The following parameter values were used to produce the final results: **n=18, m=50, j=800, k=10**.

The size evaluation was not affected by the runtime issues, therefore we were able to evaluate the maximum compression achievable using the full dendrogram. However, we were limited by the clustering library's implementation, which seemingly makes use of recursion, and therefore easily reaches Python's maximum recursion limit for graphs bigger than 100,000 nodes. Due to this, we were only able to evaluate graphs built from up to 150 traces. This evaluation was conducted in the following steps:

(1) Select a new set of *m* traces as training set
(2) Build the prefix tree from the training set
(3) Build the clustering dendrogram from the prefix tree
(4) Perform all merges from the dendrogram at once
(5) Evaluate the final size
(6) Repeat all steps from 1 *n* times

Results from this procedure were aggregated into an average compression ratio, then the entire procedure was repeated for a different value of *m*. In total, 10 different values of *m* between 50 and 150 were tested and tabulated.

5 RESULTS

This section will show and describe the results obtained from the evaluation and their limitations. It will also compare results obtained with our approach with alternative approaches tested on the same dataset.

5.1 Compression

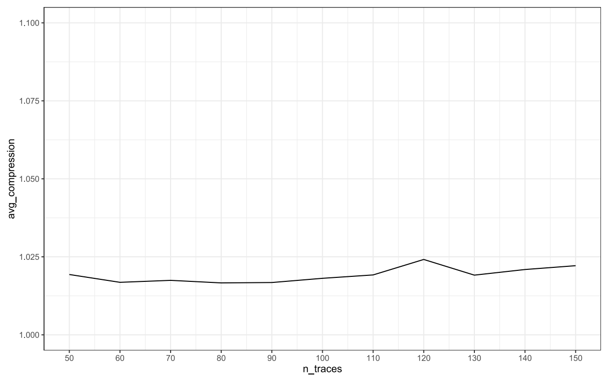

Figure 1: Average compression over number of traces

Figure 1 shows the average compression ratios for models over the number of traces used. The ratios were calculated on models from 50 to 150 traces and oscillated around 1.02, with a surprisingly low variation, oscillating between a minimum of 1.0166 for 80-trace models and a maximum of 1.024 for 120-trace models. No significant trend was present beyond what could be attributed to random differences in the training set composition. This result highlights the significant drawbacks of the mismatch between the clustering algorithm's implementation and the requirements of Finite State Machine processing. Notably, the average reductions did not seem to be affected by the increase in size of the graphs, at least within the range of sizes tested during our evaluation. The variance in reduction rates is also noticeably small, within a margin of 0.1, which seems to suggest that the ratio of dendrogram merges performed over the entire dendrogram does not vary with the size of the graph. In any case, it is clear that the graph's topology diverges quickly from the dendrogram's representation as more nodes are merged during determinization.

5.2 Specificity and Recall

Benchmarking conducted to estimate the specificity and recall of the models through the merging process did not show any meaningful change in the measured values. Due to the nature of our benchmarking setup, the initial specificity for the prefix tree was 1.0, while the initial recall was 0.0. This reflects the fact that the initial prefix tree accepts all and only the training samples, without however accepting any of the test samples, either positive or negative. The compressed models did not show any change in the measured values for specificity and recall within the scope of the testing conducted.

The specificity value remained fixed at exactly 1.0 throughout the merging process, indicating that no false positive classifications had ever taken place. This seems to suggest that our merging approach is complete, within the scope of the benchmarking conducted. This is also surprising, however, as a certain degree of error is expected by heuristic approaches such as ours when attempting to generalise a classification problem from samples.

The recall values also remained fixed at 0.0 throughout the merging. This shows that no positive trace was recognized from the test set, or, in other words, that our models do not generalize at all to unseen traces. This is possibly due to the fact that the number of merges performed was insufficient. However, it is surprising that not a single positive trace was ever recognized throughout the testing, this might suggest a flaw in one of our assumptions.

It should be once again noted that such results were obtained on limited sample sizes with limited compression ratios, due to the restrictions and limitations of the proof-of-concept implementation.

5.3 Runtime

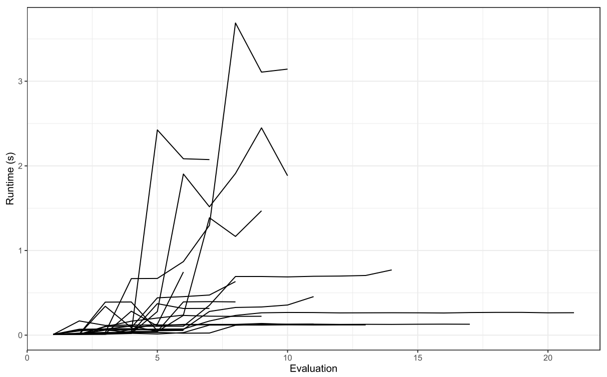

Figure 2: Runtime growth of various models during merging

Figure 2 shows the runtime growth during merging for multiple equivalent models. In contrast to the previous three evaluation criteria measurements, which showed very limited variation, the measured runtimes varied randomly by multiple orders of magnitude, making them difficult to analyse or plot meaningfully. The variations were so significant, in fact, that not all of the iterations could be completed successfully as some models' evaluation required more time than was manageable and had to be interrupted. Notably, this significant variation was measured in the evaluation of models built and compressed with the same parameters, indicating that the runtime heavily depends on the intrinsic topology of the training sample sets, even though they are generated from traces from the same system. This can be explained by the fact that the runtime is theorically expected to grow exponentially with the number of non-deterministic transitions introduced during merging. These results suggest that random differences in the topology of training sample sets have a significant and unpredictable effect on the runtime of the models' evaluation. Notably, this high variability was observed over multiple sample set sizes and number of merges performed, although both of these parameters could only be tested in limited ranges due to the issues described in the next chapter.

6 DISCUSSION OF IMPLEMENTATION CHALLENGES

During the implementation of the proof of concept program various technical and practical challenges arose. Some of these could not be fully overcome within the timeframe of the project, and therefore affected the study's results. This section will detail the nature of these challenges and their effects on the results.

6.1 Non-determinism and Exponential Growth

As previously mentioned in section 3.3, merging states in a DFA while maintaining determinism requires a specialized determinization algorithm to be executed after each merge. Such an algorithm should traverse the graph and merge equivalent nodes together whenever they are children of the same node. Each merge that such an algorithm performs can cause new instances of non-deterministic transitions to occur, cascading changes throughout the entire graph. Designing an effective determinization strategy therefore requires ensuring the ability of the algorithm to keep traversing the graph until the latter is fully deterministic.

In the case of a tree-shaped automaton, a simple recursive strategy is enough to guarantee determinism: recursively determinizing children of a node created by a merge is guaranteed to terminate once the leaf nodes are reached. However, implementing determinization in a directed graph with possible cycles poses significant additional challenges. Firstly, since in such a graph there can be a path from a node to any other, a single determinization run can, in the worst case, continue merging nodes until there is only one left in the entire graph. Secondly, the implementation of such an algorithm is difficult due to the fact that each recursive call made within a stack frame can invalidate object references used within the same frame, due to cyclical paths in the graph.

Merging determinization is an open problem in the field, as no conclusive solution was found by our research. While mentions of similar determinization algorithms were found in related literature, we could not find any stable and publicly available implementation. Verwer and De Weerdt describe the need for an analogous algorithm in [20] and imply having a working proof of concept implementation in [18]. However, they provide neither access to their implementation, nor a description or pseudocode of such an algorithm that could be used to inform an independent implementation. Due to the time constraints of this project, we were not able to develop a full determinization algorithm. We instead decided to implement the naive version previously described as sufficient for tree-shaped automata. This allowed us to partially mitigate issues related to non-determinism, but not solve them outright.

The non-deterministic nature of the generated models means that the runtime of the sample classification process increases exponentially with the number of non-deterministic transitions present in the model. While a DFA can only have one path that matches an input word, an NFA may have multiple ones, which have to be computed in parallel. In other words, every non-deterministic transition encountered during a traversal of the graph multiplies the time required to complete the traversal by the amount of possible paths. Since each merge performed during the model generation can introduce one or more non-determinism instances, the growth in evaluation runtime due to non-deterministic merges far outpaces the decrease due to the shrinking of the model. We observed this in practice during our evaluation tests, as the growing runtime made it practically impossible to evaluate models beyond a limited input size and number of merges. We attempted to mitigate this issue by using a Depth-First Search strategy for finding paths, which did marginally improve the average performance, but overall did not make a significant difference.

6.2 Merge-skipping and Information Loss

As mentioned in section 3.3, during the model generation merges from the dendrogram have to be skipped if they were affected by a previous determinization. As subsequent determinizations affect a growing share of the graph's nodes, the share of dendrogram merges applied decreases. This means that, despite the dendrogram describing merges up to a fully merged graph with one node, the number of merges applicable in our algorithm is limited, which limits the compression ratio. As a result, we were not able to utilize the full information provided by the clustering algorithm, due to a mismatch between the functionality of the clustering implementation and the requirements of ours. In practice, we observed this issue having drastic impacts on the compression of our generated models. Smaller models appeared to be more susceptible to this issue, with a compression ratio limited to 1.02 in the worst case for a 7900-node graph. This issue was compounded by the nature of our merging implementation, which did not preserve information about the identity of merged nodes, and by the nature of the clustering implementation we used, which was not designed to support only a selection of merges being performed.

7 THREATS TO VALIDITY

This section will briefly discuss the features of our study which might threaten its validity and generalizability.

7.1 External Validity

External validity refers to the generalizability of the results to different contexts. Our approach assumes that log statements do indeed encode meaningful information about the state of their origin system. We confirmed this assumption by manually analyzing the source code of the Ripple Consensus Protocol module. However, this assumption's validity entirely depends on the nature of the logging scheme used. Logging in practice is not only used for Program Execution Monitoring. Different applications of logging require different criteria for the placement and formatting of log statements, which may imply that less information about the execution state of the system can be inferred from log traces. Similar issues can arise from the variable frequency and distribution of log statements in source code. Infrequent logging may not provide enough information for the system models to be reliable, while an uneven distribution of logs in different submodules may bias the models into incorrectly estimating the topology of the system.

7.2 Internal Validity

Internal validity refers to the quality of the evaluation conducted. The main issue in this regard is the small sample sets used in the evaluation. The non-determinism issue discussed in section 6.1 meant that our evaluation was limited to single sets of less than 50 traces, which is not necessarily a large enough sample size to draw satisfying conclusions. Smaller sample sets are more affected by the natural stochastic variation in the topology of the samples, which was reflected in the fluctuations observed in the results, particularly in the runtime evaluation. Due to the nature of our approach, which was heavily affected by the natural topology of the prefix trees, the sample size used in our evaluation was not sufficient to ensure the internal validity of our results.

8 RESPONSIBLE RESEARCH

As a matter of theoretical computer science, our research did not imply significant ethical issues. However, we consider two matters worth addressing: data identifiability and reproducibility.

The log datasets we used to train the models contained information about other nodes on the XRP Network. We did not consider such data to be sensitive, since all its information is already publicly available on the XRP ledger. Additionally, XRP wallets are pseudonymous, which decreases the risk of personal identification. A case could be made that it would still be good practice to not release the training data despite the low risk of personal identification, but this has to be weighted against the need for reproducibility of the study. In any case, it should be noted that the generated models do not contain any of the dynamic data from the network, since it is filtered out by the usage of the Syntax Tree.

While computer science is not as affected by the so called "replication crisis" as the social sciences, it is still crucial for the reliability of scientific results that they be replicable. We took multiple steps during our process to ensure this replicability. The source code we used to perform all steps of the implementation, including the evaluation and data visualization, is freely accessible on a Github repository [17]. The parameters and settings used in the evaluation are mentioned in section 4.5, and the full dataset used is available at [3]. The decision to publish the dataset is in violation of the previously mentioned good practice of not publishing data which may be somewhat identifiable, however, we deemed the risk of identification from publicly available pseudonymous data already hosted on a blockchain low enough to be shadowed by the need for replicability of the study results.

9 CONCLUSIONS AND FUTURE WORK

This study was aimed at evaluating the performance of an approach to the DFA inference problem using hierarchical graph clustering, with a proof-of-concept implementation built on data from the Ripple Consensus Protocol. The performance evaluation was based on four metrics: size reduction, specificity, recall, and runtime. The implementation incurred into issues which highlighted the challenges posed by this approach, some of these issues could not be resolved within the timeframe of the project, and affected the results. As a result, the evaluation was limited in scope, and constrained to small sample sizes and incomplete model generation.

Results obtained from the evaluation are not promising: while the computational cost of using the generated models increases exponentially, they do not seem able to generalize to unseen data, and their size reduction is limited. However, the limitations of the evaluation mean that such results are to be considered inconclusive. The implementation challenges highlighted during this project have to be tackled first, before conclusive results can be achieved about the efficacy of this approach.

Firstly, a reliable algorithm must be developed for determinizing a DFA after merging states, with support for cyclical DFAs. Another challenge is better integration of the hierarchical clustering with the DFA requirements. If possible, a custom clustering algorithm should be implemented which natively maintains determinism in its clusters. Such an algorithm would likely require a different implementation of the Graph data structure, which should also be improved. Finally, we have not properly tested the assumption that relative empirical transition frequencies provide a good measure for the relatedness of DFA states. Further research should test this assumption and either confirm it, or design better measures for estimating the relatedness of neighboring DFA states.

REFERENCES

[1] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (Oct 2008), P10008. https://doi.org/10.1088/1742-5468/2008/10/p10008

[2] Thomas Bonald, Nathan de Lara, Quentin Lutz, and Bertrand Charpentier. 2020. Scikit-network: Graph Analysis in Python. *Journal of Machine Learning Research* 21, 185 (2020), 1–6. http://jmlr.org/papers/v21/20-412.html

[3] Tommaso Brandirali, Calin Georgescu, Pandelis Symeonidis, and Thomas Werthenbach. 2021. *XRP Ledger Consensus Protocol Debug-level Log Traces.* https://doi.org/10.5281/zenodo.5035325

[4] Rafael C Carrasco and Jose Oncina. 1994. Learning stochastic regular grammars by means of a state merging method. In *International Colloquium on Grammatical Inference*. Springer, 139–152.

[5] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.

[6] Hetong Dai, Heng Li, Che Shao Chen, Weiyi Shang, and Tse-Hsun Chen. 2020. Logram: Efficient log parsing using n-gram dictionaries. *IEEE Transactions on Software Engineering* (2020).

[7] Santo Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3-5 (2010), 75–174.

[8] E Mark Gold. 1978. Complexity of automaton identification from given data. *Information and control* 37, 3 (1978), 302–320.

[9] Marijn JH Heule and Sicco Verwer. 2013. Software model synthesis using satisfiability solvers. *Empirical Software Engineering* 18, 4 (2013), 825–856.

[10] Edward F Moore. 2016. Gedanken-experiments on sequential machines. In *Automata Studies.(AM-34), Volume 34*. Princeton University Press, 129–154.

[11] Leonard Pitt and Manfred K Warmuth. 1993. The minimum consistent DFA problem cannot be approximated within any polynomial. *Journal of the ACM (JACM)* 40, 1 (1993), 95–142.

[12] Bernhard Plattner and Juerg Nievergelt. 1981. Special feature: Monitoring program execution: A survey. *Computer* 14, 11 (1981), 76–93.

[13] Jared Rand and Andriy Miranskyy. 2021. On Automatic Parsing of Log Records. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 41–45.

[14] Marijn Roelvink, Mitchell Olsthoorn, and Annibale Panichella. 2020. Log inference on the Ripple Protocol: testing the system with an empirical approach. (2020).

[15] David Schwartz, Noah Youngs, Arthur Britto, et al. 2014. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper* 5, 8 (2014), 151.

[16] Sameer Shende. 1999. Profiling and tracing in linux. In *Proceedings of the Extreme Linux Workshop*, Vol. 2. Citeseer.

[17] TommasoBrandirali, Mitchell Olsthoorn, and Annibale Panichella. 2021. *TommasoBrandirali/WhatTheLog: 1.0.* https://doi.org/10.5281/zenodo.5035298

[18] SE Verwer, MM De Weerdt, and Cees Witteveen. 2007. An algorithm for learning real-time automata. In *Benelearn 2007: Proceedings of the Annual Machine Learning Conference of Belgium and the Netherlands, Amsterdam, The Netherlands, 14-15 May 2007*.

[19] Sicco Verwer and Christian A. Hammerschmidt. 2017. flexfringe: A Passive Automaton Learning Package. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 638–642. https://doi.org/10.1109/ICSME.2017.58

[20] Sicco E Verwer, Mathijs M De Weerdt, and Cees Witteveen. 2006. Identifying an automaton model for timed data. In *Benelearn 2006: Proceedings of the 15th Annual Machine Learning Conference of Belgium and the Netherlands, Ghent, Belgium, 11-12 May 2006*.