Inferring Log-Based Behavioural System Models using Markov Chains

Thomas Werthenbach
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

An essential step of software development is obtaining an understanding of the behaviour of a system. Accurate state models of system behaviour might help software developers build such an understanding. There exist several techniques for automatically inferring models on system behaviour using log analysis, but these do not scale well for systems that produce large amounts of logs.

In this study, we present an approach for inferring concise state models of system behaviour using log analysis, called MASM (Markov Algorithm for inferring concise State Models), which implements a Markov chain. We argue this approach has the potential for higher scalability than existing techniques, as it attempts to approximate a state model by exploiting the properties of Markov chains. This is achieved by considering a sequence of log statements as a stochastic process, which enables MASM to infer a naive state model from a set of log statements, which is then minimized using the properties of Markov chains.

We evaluated MASM in an empirical study on the XRP Ledger Consensus Protocol. In this empirical study, MASM was evaluated on accuracy at different compression rates, and scalability. The results indicate that MASM shows several knee points in terms of accuracy, suggesting several optimal compression rates. We also found that the run-time of MASM scales linearly with the size of the dataset. Finally, we discovered that MASM is unable to outperform a random compression algorithm, which compresses the model by randomly merging states, in terms of accuracy. However, further research is required on measuring the readability of the produced models to derive definitive conclusions on the performance of MASM compared to a random compression algorithm.

1 INTRODUCTION

An essential step of software development is obtaining an understanding of the behaviour of a system, which is required for tasks such as test case generation [4, 10] and the analysis of software processes [6]. Accurate models of system behaviour might help software developers build such understanding and "reduce software development effort and improve quality" [22].

There exist several techniques for inferring models on system behaviour, such as profiling [11], tracing [20], or source code analysis [16]. These techniques perform well enough for relatively small systems [17], of which the source code is available, but suffer from limitations. As such, profiling and tracing introduce additional overhead, making this technique infeasible for large real-time systems in which performance is essential. The aforementioned methods also require full access to the source code of the given system and any third-party packages that the system might depend upon. Such access might not always be available. For these reasons, other approaches need to be considered.

This paper focuses on modelling system behaviour using log analysis, as logs can be retrieved without introducing additional overhead nor requiring access to the source code. Therefore, systems that already produce logs do not need to be adapted, nor would their performance decrease by utilizing such a technique.

Various studies have used log analysis to infer state models of system behaviour [13, 17, 21], which have shown promising results. However, as the problem of constructing a minimal state model is NP-hard [1], existing solutions are unable to perform this task for large and complex systems within a feasible amount of time. For this reason, further research is required on methods that have the potential for higher scalability.

This study investigates the effectiveness of using Markov chains for inferring accurate and concise state models of system behaviour from a set of logs. A Markov chain is a prevalent mathematical modelling technique for stochastic processes [28]. By considering a sequence of log statements as a stochastic process, we can apply the Markov chain model on such sequences, using the Markov chain's adjacency matrix to keep track of the probability of each possible state transition, where the states in the state model correspond to one or more log statements.

Our approach, called MASM (Markov Algorithm for inferring concise State Models), attempts to infer concise state models of system behaviour in several steps. We first infer a naive initial state model from a large dataset of logs. While doing so, a Markov chain is trained such that its adjacency matrix contains the approximate probabilities of the state transitions. After the initial model has been constructed, MASM attempts to compress this model by exploiting certain properties of adjacency matrices. One such property is that two states in the model can be merged when they correspond to equivalent rows in the adjacency matrix, as they share the same probabilities for outgoing edges. MASM compresses the initial model until either a maximum result size or minimum accuracy value is met, which are set by the user. Some compression iterations likely result in several sets of candidate states to merge. In such cases, all sets are evaluated to determine which would result in the best model according to the user's priorities.

We argue that MASM has a potential for higher scalability than existing techniques in inferring a state model on system behaviour, as we approximate a behavioural state model by exploiting the properties of Markov chains.

We evaluated MASM in an empirical study on the logs of the XRP Ledger Consensus Protocol by Ripple [3]. The results indicate that MASM shows several knee points in terms of accuracy, suggesting several optimal compression rates [27]. For the logs of the Consensus Protocol, these knee points are located at the compression rates of 52%, 69%, and 78%. We also found that MASM is unable to outperform a random compression algorithm, which compresses the model by randomly merging states, in terms of accuracy. However, further research is required on measuring the readability of the produced models to derive definitive conclusions on the performance of MASM compared to a random compression algorithm. Lastly, we discovered that MASM scales linearly with the size of the training set.

This paper is structured as follows: first, Section 2 provides a brief overview of the required background knowledge. In Section 3, a detailed description of MASM is presented. An overview of the empirical experiment performed in this study can be found in Section 4. Furthermore, the results can be viewed in Section 5, followed by a brief description of the threats to validity in Section 6. Section 7 contains more information about the ethical aspects of this study. Finally, Section 8 highlights the conclusions of this research and future work.

2 BACKGROUND

2.1 Terms and definitions

This section provides descriptions of the main concepts and terms relevant to this study.

Log statements: We define log statements as a textual representation of occurred events. Several examples of log statements can be found in appendix A. In this study, log statements start with a timestamp of the event. This timestamp is followed by a message consisting of a static part, which can be used as an identifier for the event type, and a dynamic part, which contains some of the event parameters [12]. For instance, the event type of the log statement "`2020-10-12 18:30:29.54726 agent_3` **`votes`** `YES` **`on`** `TX_5`" can be recognized by the static part **`votes`** and **`on`**. The remaining parts of the log statement are dynamic and may vary.

Log trace: A log trace is a list of logs statements. Log traces are formally defined as $\langle l_1, ..., l_k \rangle, \forall i_{1 \le i \le k} : l_i \in \mathcal{L}$, where $\mathcal{L}$ is the set of all possible log statements.

Finite state machines: Log traces consisting of log statements are parsed by MASM to infer a finite state machine. The definition of these finite state machines is similar to that of *finite automata*, as defined by Sipser [18]. Such finite automata are be represented by a tuple $(Q, \Sigma, \delta, q_0, \mathcal{F})$, where Q is the set of states, Σ is the alphabet (set of log statements), δ is the transition function $Q \times \Sigma \longrightarrow Q$, q_0 is the start state or root, and $\mathcal{F}$ is the set of accept states.

In this study, we apply a variation on the formal definition of finite automata. More specifically, the finite state machines used in this study contain one more element, such that they can be represented by the tuple $(Q, \Sigma, \delta, q_0, \mathcal{F}, \mathcal{P}_s)$, where $\mathcal{P}_s$ is defined as a trained set of all probabilities for transitions between any states, when the next log template is unknown, such that $\mathcal{P}_s = \{\langle a, b, p \rangle, \forall a, b \in Q \wedge 0 \le p \le 1\}$. Furthermore, all edges that have a probability $p = 0$ are discarded from the model. More formally, $\forall \langle a, w, b \rangle \in \delta : \{a, b \in Q \wedge w \in \Sigma\} \Leftrightarrow \nexists \langle a, b, p \rangle \in \mathcal{P}_s : \{p = 0\}$.

Markov chains: MASM implements the usage of Markov chains to infer finite state machines. A Markov chain is a relatively simple and prevalent data structure used to analyze and model stochastic processes [28]. As such, applications adopting Markov chains have been proposed in various studies. For example, Sahin and Sen have used Markov chains to predict wind speed states [14]. Markov chains have also been used for predicting train states and estimating delay probabilities [28].

As Markov chains are used to model stochastic processes, we consider a sequence of log statements, i.e. a log trace, as a stochastic process, allowing for the usage of Markov chains.

We define a Markov chain such that it can be represented by an adjacency matrix M, in which the number of rows and columns are equal and finite, such that the adjacency matrix has a size of $n \times n$, where $n \in \mathbb{N}$. An adjacency matrix defines the probabilities of all possible state transitions. The rows of the adjacency matrix represent the current state of a system, and the columns represent the next state of a system. Formally, given a function $\mathcal{A}$ which maps indices in the adjacency matrix to states in the finite state machine, the adjacency matrix contains the probability of edge transitions such that $\forall i_{1 \le i \le n}, j_{1 \le j \le n} : (M_{i,j} = p \wedge \exists \langle \mathcal{A}(i), \mathcal{A}(j), p \rangle \in \mathcal{P}_s \wedge 0 \le p \le 1)$, where $\mathcal{P}_s$ is the set of probabilities for state transition as defined above. A defining characteristic of Markov chains is that the probability of transitioning to any state depends solely on the current state. Earlier states do not affect the next state transition.

The ability of adjacency matrices to be compressed was exploited during this study. Such compressions can be performed without introducing error using row and column equivalence [19]. More specifically, when the rows in the adjacency matrix of some state A and some other state B are equivalent, these states can be compressed into a state AB, as they contain equivalent probabilities for all outgoing edges. For this study, the state AB represents the occurrence of either the event type of state A or B.

2.2 Related work

An example of an algorithm similar to MASM is flexfringe [21]. Flexfringe is a greedy algorithm that infers state models of system behaviour. The main aim of flexfringe is to reverse-engineer these state models using log analysis and "accelerate and improve the software development and inform domain experts about the processes actually executed in a system" [21].

Flexfringe's effectiveness was investigated in a study by Roelvink [13], in which flexfringe was used to infer an empirical model of the system behaviour of the Consensus Protocol of the XRP Ledger. This model was compared to a theoretical model of the Consensus Protocol to verify whether the system behaves as expected.

Furthermore, Shin et al. [17] presented a method that used a *divide and conquer* approach to infer a model on the system behaviour, called SCALER. They approach this problem by first inferring models of individual components of a system using log analysis, which are then combined into a system-level model.

As the problem of generating a minimal finite state machine is NP-hard [1], existing solutions are unable to perform this task for large and complex systems within a feasible amount of time. For this reason, further research is required on methods that have the potential for higher scalability.

3 APPROACH

This section describes in detail the manner in which MASM attempts to infer concise state models using log-analysis.

To summarize, we first generate a naive initial finite state machine that has been inferred from parsing log statements. To parse log statements, we implemented a data structure capable of recognizing the event types of log statements, called a *syntax tree*, which recognizes log statements based on their static part. The *syntax tree* is formally defined in Section 3.1. While the initial state machine is being inferred, the probabilities of all state transitions are stored in the Markov chain's adjacency matrix, which is described in Sections 3.2 and 3.3.

Once the initial state machine has been inferred, MASM compresses this state machine iteratively until either a specified maximum result size or minimum accuracy value is reached, which are both specified by the user. The compression is performed by exploiting properties of adjacency matrices, which is elaborated in Section 3.4. As there may exist several ways to compress the initial model, we require the ability to determine the best option. This process is described in Section 3.5.

3.1 Syntax tree

The *syntax tree* is used by MASM to parse log statements by recognizing their static parts. We achieve this by applying regular expressions. Formally, a *syntax tree* can be represented by a tuple $(\mathcal{N}, r, \phi, \mathcal{F}_s)$, where $\mathcal{N}$ is the set of all nodes in the tree, r is the root node, ϕ is the set of all edges, and $\mathcal{F}_s$ is the set of all accepting paths, such that $\mathcal{F}_s = \{\langle n_1, ..., n_k\rangle, n_1 = r \wedge \nexists\langle n_k, n_{k+1}\rangle \in \phi \wedge \forall i_{1\leq i\leq k-1} : \{n_i \in \mathcal{N} \wedge \langle n_i, n_{i+1}\rangle \in \phi\}\}$. Each node $n \in \mathcal{N}$ has a name for the specific event type and a regular expression capable of recognizing a part of a log statement. More specifically, $\forall\langle l_1, ..., l_k\rangle \in \mathcal{L} : \exists\langle n_1, ..., n_k\rangle \in \mathcal{F}_s$, where $\mathcal{L}$ is the set of all log statements as defined in Section 2.1 and the regular expression of n_i matches l_i, for all $i_{1\leq i\leq k}$. It is important to note that the *syntax tree* requires to be manually constructed to recognize the event types, and that we assume that all unique event types have a unique static part which can be used to recognize the event types. The *syntax tree* is not directly part of the resulting model, as its only purpose is to parse log statements.

3.2 State model representation

To infer a concise model, a larger initial model needs to be created. This initial model is created using all the log traces in a training set. For this study, two state model representations were considered, *prefix trees* and *unique state graphs.*

3.2.1 *Prefix tree.*

A *prefix tree* is a data structure in which all log traces are represented without any loops, making it an accurate reflection of the dataset used to infer it. A *prefix tree* is characterized by its determinism, which means that an arbitrary state can not have two outgoing states corresponding to the same event type.

When this functionality was first implemented, we discovered that MASM could not compress the initial state model to a more concise state model using a *prefix tree* in a reasonable amount of time. This is mainly caused by the manner in which MASM searches for options to compress the initial model and by the size of the initial model: usually more than one million states, depending on the size of the dataset. We therefore decided to abandon the *prefix tree* representation for the remainder of this study.

3.2.2 *Unique state graph.*

This state model representation aims to decrease the size of the initial model, allowing MASM to infer a concise state model in a shorter amount of time. A *unique state graph* differs from a *prefix tree* in that it can only contain a single state for every event type.

A *unique state graph* can be formally defined by $(Q_u, r_u, \Sigma_u, \delta_u, f_u)$, where Q_u is the set of all states, r_u is the root of the *unique state graph*, Σ_u is the alphabet, equal to the set of all log statements $\mathcal{L}$, δ_u is the set of all edges, and f_u is the accepting state.

The *unique state graph* contains a path from the root r_u to the accepting state f_u for every log trace in the training set. Formally, given the set $\mathcal{T}$, which contains all log traces the *unique state graph* was trained upon, and a function L which maps log statements to their corresponding states, we define this characteristic as $\{\forall\langle l_1, ..., l_k\rangle \in \mathcal{T}, \forall i_{1\leq i\leq k-1}\} : \{\exists\langle L(l_i), L(l_{i+1})\rangle \in \delta_u \wedge L(l_k) = f_u\}$. We say that a log trace $\langle l_1, ..., l_k\rangle$, which is not in $\mathcal{T}$, is accepted when $\forall i_{1\leq i\leq k-1} : \exists\langle L(l_i), L(l_{i+1})\rangle \in \delta_p \wedge L(l_k) = f_p$.

The *unique state graph*'s distinguishing characteristic that it can only contain a single state for every event type, is defined such that $\forall x, y \in \mathcal{L} : (L(x) = L(y) \Leftrightarrow \mathcal{S}(x) = \mathcal{S}(y))$, where $\mathcal{S}$ is a function capable of recognizing a log statement's event type using the *syntax tree.* This characteristic enforces the constraint $|Q_u| \leq |\mathcal{F}_s|$, where $\mathcal{F}_s$ is the set of all accepting paths of a *syntax tree*, as defined above. This constraint is likely to improve the performance of MASM in terms of run-time significantly.

A disadvantage of utilizing a *unique state graph* compared to a *prefix tree* is that the *unique state graph* can enforce fewer constraints than the *prefix tree.* For instance, if there exists a constraint specifying that a certain event type must occur a fixed amount of times in a row, the *prefix tree* can detect log traces where this constraint is not satisfied, while a *unique state graph* is unable to detect such violations. Another disadvantage of using a *unique state graph* is that it contains more paths than a *prefix tree*, limiting its ability to detect invalid log traces.

3.3 Training the Markov chain

To create the initial model, the adjacency matrix needs to be trained with existing log traces. This is done by following these steps:

(1) Initialize an adjacency matrix of $|\mathcal{F}_s| \times |\mathcal{F}_s|$ with a value of 0 for every cell, where $\mathcal{F}_s$ is the set of all accepting paths in the *syntax tree.*
(2) Loop over all the log statements of the log traces of the training set. Add a value of 1 to the corresponding cell in the adjacency matrix for every log statement transition.
(3) Remove unreachable states from the adjacency matrix.
(4) Normalize every row by dividing the value of every cell with the sum of the row.

3.4 Merging states

After the initial model has been inferred from log traces, it will likely not yet meet the specified requirements of either a maximum result size or minimum accuracy, which the user can specify. To compress the initial model to a more concise model, in which one of these requirements is satisfied, MASM implements three methods to find candidate states to merge:

(1) Search for subsequent states in which the previous state has a probability of 1 to go to another state. Formally, given the set of probabilities for all state transitions $\mathcal{P}_s$ and the set of all edges δ_p, a function S, which can detect these subsequent states, is defined in equation 1.

$$S(A, B) \rightarrow \langle A, B \rangle \in \delta_p \wedge \langle A, B, p \rangle \in \mathcal{P}_s \wedge p = 1 \quad (1)$$

As a probability of 1 indicates that the next event type always follows the previous event type, they can be considered the same state and be merged. The new state should have a self-loop, as the merged sequence should still be accepted.

(2) Search for duplicate rows in the adjacency matrix. This method was inspired from an article by Spears [19], in which it is proven that probabilistic matrices can be compressed with the smallest amount of error by merging the most similar states.

(3) Search for duplicate columns in the adjacency matrix. This method was inspired by [19] as well.

It is important to note that there exists a dynamic threshold for all of the aforementioned methods. This threshold indicates the extent to which two states may diverge from the requirements to be classified as candidate states. A formal definition is given in equation 2, in which a function F takes two rows and a threshold ε and determines whether the rows are equivalent.

$$F(A, B, \varepsilon) \rightarrow A = B \Leftrightarrow |A| = |B| \wedge \forall i_{0 \leq i < |A|} : (|A_i - B_i| \leq \varepsilon) \quad (2)$$

The dynamic threshold ε will be increased when no candidate states are found.

When the two states $q_i, q_j \in Q_u$ are merged, the adjacency matrix needs to be modified. This is done according to an adaptation of the guidelines described in [19]:

(1) Sum the probabilities of the rows corresponding to q_i and q_j and place them in the row of q_i.
(2) Divide all the values in the row of q_i by 2, as the sum of the values in the row is now 2 and needs to be normalized.
(3) Delete the row of q_j.
(4) Sum the probabilities of the columns corresponding to the two states and place them in the column of q_i. All the edges directed towards q_j are then directed towards q_i.
(5) Delete the column of q_j.

According to these guidelines, merged states adhere to an OR condition, which entails that only one of the event types corresponding to a state, constructed by merging individual states, needs to occur for transitioning to that specific state.

3.5 Evaluating a model

The initial model is the most accurate, as it is the closest to a reflection of the log traces that were trained upon. Therefore, this initial model will contain implicit constraints, such as state X must be followed by state Y. However, as the loss of accuracy is a fundamental characteristic of compressing such a model, these implicit constraints can be lost in the process. For instance, if the arbitrary pair of subsequent states $q_a, q_b \in Q$, such that $\langle q_a, q_b \rangle \in \delta_u$, are merged into a new state q_{ab}, there exists no constraint on the order in which the log statements corresponding to q_a or q_b can appear subsequently in a log trace.

Table 1: Overview on the result classes

	True	**False**
Positive	Valid traces recognized as valid. (TP)	Invalid traces recognized as valid. (FP)
Negative	Invalid traces recognized as invalid. (TN)	Valid traces recognized as invalid. (FN)

The aforementioned methods to find the best candidate states to merge attempt to minimize the loss of accuracy while compressing the model. These methods may however regularly result in several sets of candidate states. Therefore, MASM must be able to evaluate all candidates to determine which candidate to merge. We achieve this by temporarily merging these candidate states and evaluating the resulting model.

When evaluating a single trace, the result will be true/false positive/negative, depending on the input and whether the model recognizes it. These result classes are further elaborated in Table 1.

For this study, 4 metrics were used for evaluation:

- Recall: specifies how many of the valid traces are recognized as valid.

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

- Precision: specifies how many of all the traces recognized as valid are actually valid.

$$Precision = \frac{TP}{TP + FP} \quad (4)$$

- Specificity: specifies how many of the invalid traces are recognized as invalid.

$$Specificity = \frac{TN}{TN + FP} \quad (5)$$

- Conciseness: specifies the degree of compression.

We chose these metrics because they are well-established, often used in literature [8, 15, 17], and result in normalized values on the model's accuracy, which simplifies comparison.

While the precision metric is not used for evaluating candidate states, as we argue that the model's behaviour is sufficiently captured by specificity and recall, the precision was included in the results of the empirical study.

To determine what candidate states should be merged, we calculate a score for all sets of candidates (in equation 6). The candidate states of the set that obtains the highest score will be merged.

$$Score = w_c \cdot \frac{amount_{current}}{amount_{max}} + w_a \cdot \frac{specificity + recall}{2} \quad (6)$$

$amount_{current}$ is this number of states that would be merged for the current set of candidate states. $amount_{max}$ is the maximum number of states that would be merged for all sets of candidate states. The weights of conciseness w_c and accuracy w_a can be specified by the user and may vary depending on their priorities.

One of the main reasons we decided to abandon the *prefix tree* state model representation is the manner in which this score is calculated. The calculation of the specificity and recall metrics requires a large number of log traces to obtain an accurate result. The usage of larger models, such as a *prefix tree*, would greatly increase the number of times this calculation needs to be performed, which causes unreasonably long execution times. It would also require a larger set of valid and invalid log traces to guarantee coverage on all paths in the *prefix tree.*

4 EMPIRICAL STUDY

The purpose of this section is to describe how MASM was used to evaluate the performance of a Markov chain-based algorithm. First of all, Section 4.1 states the research questions this experimental study aims to answer. Section 4.2 explains the benchmark used during this study, and Section 4.3 provides details on the experimental setup. This includes the methods and parameters used to prepare and run the evaluation. Finally, Section 4.4 indicates how the results from the evaluation were analyzed.

4.1 Research questions

The main research question this study aims to answer is as follows:

How effective are Markov chains for inferring a concise and accurate state model of code behaviour?

Several aspects are considered relevant to formulate a proper answer to the stated research question. These aspects include comparison with other approaches, scalability, and the relation between conciseness and accuracy. We derived the following sub-question to clarify these aspects:

(1) *How does MASM perform in terms of accuracy over different inferred model sizes?*
(2) *How does MASM scale with the size of the training set?*
(3) *How does MASM perform compared to other approaches studied by peers?*
(4) *How does MASM compare to a random compression algorithm?*

4.2 Benchmark

This empirical study is a case study on the logs of the XRP Ledger. The XRP Ledger is a distributed payment system for the XRP cryptocurrency. More specifically, this empirical study focuses on the Consensus Protocol of the XRP Ledger, which aims to achieve consensus on which nodes are members of the network and which transactions should be included in the next block [3]. Since the XRP Ledger produces large amounts of logs, it is capable of creating datasets of sufficient size for this study.

The logs of the XRP Ledger were first filtered to remove all log statements unrelated to the Consensus Protocol. The resulting logs were then divided into several log traces. As the Consensus Protocol works with consensus rounds, of which the start and end are clearly indicated with log statements, we split the filtered logs into individual traces. These traces were used to create 5 different datasets of 1000 log traces each. Each log trace contains 995 log statements on average.

4.3 Experimental setup

MASM requires certain parameters as input. Due to time constraints, only a single set of experimental parameters was used for the evaluation. These parameters can be found in Table 2. Furthermore, The hardware used to evaluate MASM is the HP ZBook Studio G5. This machine contains a 12 core Intel i7-8750H CPU @ 2.20GHz with 16 GB of Random Access Memory and runs Microsoft Windows 10. The used Python interpreter is of version 3.9.

Table 2: Experimental parameters

Parameter	Value
Conciseness weight w_c	0.5
Accuracy weight w_a	0.5

4.3.1 Accuracy: To evaluate the accuracy of MASM on a variety of sizes, we adapted MASM to store the value of the metrics specificity, precision, and recall, after each individual compression. To ensure the generation of samples for all sizes, the input parameters of MASM were configured such that the initial model would be compressed iteratively to a one-node model, i.e. a model consisting of a single state, while evaluating all intermediate models.

Furthermore, we used k-folds cross-validation to evaluate the performance of MASM . This method of evaluation is often used in literature [9, 17, 26]. Each of the five different datasets was divided into five folds, allowing for a 4:1 ratio of the training and test set. Each fold was used one time as a test set, while the remaining folds were used as the training set.

To measure the specificity and precision, we require invalid log traces; these will be called negative log traces and will be created using an adapted method inspired by [13, 17, 23]. It entails that negative traces are created by randomly selecting valid traces and performing a random number of mutations on them. These mutations are not performed on individual lines, but rather on sections of subsequent lines that correspond to the same event type. The different types of mutations are: swapping two consecutive sections, swapping two randomly chosen sections, and deleting a random section. The number of applied mutations is randomly chosen in the range of one to three. We extended this implementation with additional functionality, checking whether mutated log traces are invalid using a *prefix tree* inferred from the training set. More mutations are applied when a mutated log trace remains valid.

The described accuracy experiments were also performed on a random compression algorithm. This random compression algorithm functions equivalent to MASM , except for always choosing random candidate states to merge. As there will only be a single set of candidates, the initial model will be compressed iteratively by merging randomly chosen states. We used this random compression algorithm as a baseline for this empirical study.

4.3.2 Run-time: To evaluate the scalability of MASM , experiments on the run-time were performed by varying the size of the used datasets. The dataset sizes that were evaluated upon are *100, 250, 500, 750* and *1000.* For each of these sizes, the input parameters of MASM were configured to compress the model iteratively to a one-node model 10 times. The used ratio of the training and test datasets is 1:1, as accuracy is not relevant during this experiment.

4.4 Analysis methodology

The collected data are visualized in the form of graphs on both accuracy and run-time of MASM . The graphs on accuracy (Figures 1 and 2) contain data on specificity, recall, and precision against different rates of compression. By considering the compression as

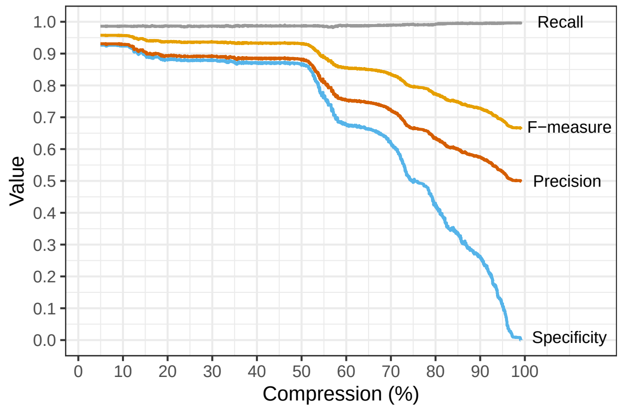


Figure 1: MASM accuracy. 75 moving point average of the metrics specificity, precision, F-measure and recall against the rate of compression (%). This graph was created using a sample size of N=5394.

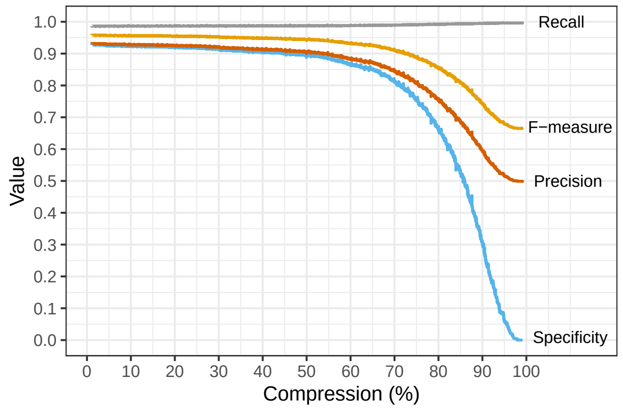


Figure 2: Random compression algorithm accuracy. 75 moving point average of the metrics specificity, precision, F-measure and recall against the rate of compression (%). This graph was created using a sample size of N=7601.

a percentage of the original size, we compensate for the differences in the size of the initial models. As both the precision and recall are about the number of true positives a resulting model can detect, these metrics can be combined. This is done using the F-measure [7], as calculated in equation 7 and displayed in Figures 1 and 2.

$$\text{F-measure} = \frac{2 \cdot recall \cdot precision}{recall + precision} \tag{7}$$

The graph on run-time (Figure 3) contains the amount of time it took to compress the model to a one-node model in seconds against the amount of log traces as the size of the dataset.

After these graphs have been constructed, they are manually inspected to find knee points [2]. These knee points indicate an optimal rate of compression, as the improvement in either one of the aforementioned metrics or conciseness will result in a significant loss of the other objective(s) [2, 27].

Using the collected data and the found knee points, conclusions can be derived to each of the sub-questions of this research. Each of the formulated sub-question are answered individually.

Finally, two models at the rates of 90% and 95% compression were visualized to give the reader an idea of the models MASM is able to infer. The produced graphs can be found in appendix B.

5 RESULTS

This section presents the results from the empirical study performed using MASM to investigate the effectiveness of utilizing Markov chains in modelling system behaviour. Section 5.1 answers the sub-questions of this empirical study. The main research question is answered in Section 5.2.

The experiments on accuracy were focused on collecting data on the specificity, precision, recall, and F-measure at different rates of compression. An equivalent experiment was performed on a random compression algorithm to provide a baseline for comparison. Finally, the run-time was evaluated to collect data on MASM 's scalability. During the latter experiment, we collected data on the amount of seconds required to infer and compress an initial model to a one-node model.

5.1 Sub-questions

5.1.1 How does MASM perform in terms of accuracy over different inferred model sizes? The results for all metrics, including the F-measure, can be found in Figure 1. Note that all the metrics have a minimum value of 0 and maximum value of 1, 0 being a complete lack of this metric and 1 being the optimal score. These metric were plotted against the rate of compression, where 0% is no compression, i.e. the initial model, and 100% represents a one-node model.

Specificity reaches a value of 0 at 100% compression. This is caused by the model's incapability to detect true negatives when the model is compressed to a one-node model. Furthermore, precision converges to a value of exactly 0.5. This can be explained because a model consisting of a single node is unable to reject any negative traces but does accept all positive traces. As the ratio of positive and negative traces is 1:1, a precision value of 0.5 follows logically.

Before the point of 52% compression, MASM achieves a very high score on all metrics. After this point, the specificity, precision, and the f-measure start to decrease as the model gets compressed. It is likely that MASM is unable to find candidate states with a low threshold ε, as defined in Section 3.4, at this specific compression rate. Therefore, a trade-off occurs: we sacrifice accuracy to gain conciseness. These trade-offs are made until the model reaches a size of a single node. Another notable result is that MASM achieves a recall close to 1 at every compression rate. This result can be explained by the usage of *unique state graphs*, as *unique state graphs* generally contain more paths than *prefix trees*.

From the accuracy results in Figure 1, one can see a few knee points. The clearest knee point is at around 52% compression. Before the 52% compression, all metrics score greater than 0.85. Another knee point can be found at around 69%. The last knee point can be found at 78% compression. It ought to be noted that these knee points differ per dataset and that the knee points found in this study are only valid for the Consensus Protocol of the XRP Ledger.

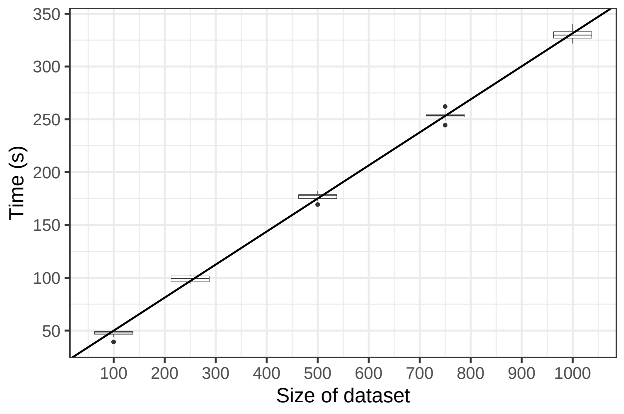


Figure 3: Boxplot graph which shows the connection between the size of the dataset and the required time to compress the initial model to a one-node model. N=50

As multiple knee points emerge from the results of this empirical study, there is no "best" rate of compression for this specific dataset. There may however be a certain rate of compression that best fits one's needs. If one would require a highly accurate model, they might choose a compression rate below 50%. On the other hand, if one requires a concise model, they might choose a compression rate of at least 80%.

5.1.2 How does MASM scale with the size of the training set? To analyze the scalability of MASM , we measured the amount of seconds required to compress this model on different dataset sizes. The dataset sizes on which this experiment was performed were *100, 250, 500, 750* and *1000*. Each of these dataset sizes was evaluated 10 times. The results can be viewed in Figure 3.

From the collected data, it becomes clear that the run-time scales linearly with the size of the dataset, which is promising for the scalability of MASM . It takes around 50 seconds to infer a model on a dataset with a size of 100 traces, while taking 325 seconds to infer a model on a dataset consisting of 1000 log traces. However, it was discovered that the largest portion of time was spent on training the Markov chain and generating the positive and negative log traces to evaluate upon. This is because all evaluation traces are loaded into memory and pre-processed, such that the individual log statements are mapped to leaves in the *syntax tree*, eliminating the need of regular expressions during compression.

The results show that using a dataset of 1000 traces takes on average less than five minutes, which might be fast enough for most appliances. However, these results are only valid to this specific dataset and hardware, and the produced models may not be sufficiently accurate for complex systems with a large number of distinct event types or short log traces.

5.1.3 How does MASM perform compared to other approaches studied by peers? Another method to infer concise and accurate state models of system behaviour using log analysis was considered by a peer in parallel research, which includes *Multiple Objective Meta-heuristic Search* (*MOMS*) [5]. As this method does not allow for the same type of evaluation, the advantages and disadvantages of *MOMS* compared to MASM are discussed.

Preliminary research shows that one of the main advantages of this *MOMS* is that *MOMS* takes less time to produce results than MASM . Another advantage of *MOMS* over MASM is that *MOMS* produces several trade-offs from which an analyst can choose the option that best fits their needs, whereas MASM requires a user to provide stopping criteria, i.e. maximum result size and minimum accuracy, to determine when to stop compressing the model, and only produces a single result. However, being able to provide these criteria allows a user to obtain full control of the compression process and produce models of the desired size and accuracy.

5.1.4 How does MASM compare to a random compression algorithm? An equivalent accuracy experiment, as was used to answer the research question of Section 5.1.1, was performed on a random compression algorithm, see Figure 2.

From this figure, it becomes clear that the random compression algorithm scores better than MASM overall on all metrics, which was an unanticipated result. We argue this is caused by the method used to generate negative traces. These are generated by applying a small number of mutations which affect the "normal flow" of the Consensus Protocol. As MASM adapts the "normal flow" of the graph as well, the model is less capable of detecting true negatives, thus explaining the loss of specificity. However, as the random compression algorithm randomly merges two states, it is less likely to alter the "normal flow" of the graph. The random compression algorithm is more likely to create new paths which are not covered by negative traces. MASM could outperform the random compression algorithm if more mutations were applied during the generation of negative traces, but further research is required.

Note that the high accuracy does not necessarily imply high readability. It is likely that MASM produces models which have higher readability than the models inferred by the random compression algorithm. Measuring a state model's readability is, however, not within the scope of this study.

5.2 Research question

How effective are Markov chains for inferring a concise and accurate state model of code behaviour? During this empirical study, we found that a Markov chain-based algorithm for inferring concise state models of code behaviour shows several knee points in terms of accuracy when evaluated on the XRP Ledger Consensus Protocol dataset. Moreover, we found that MASM does not perform better than a random compression algorithm, which randomly merges states to compress the model, in terms of accuracy. We hypothesise this is caused by the manner in which negative traces are generated, but further research is required.

Furthermore, we discovered that MASM scales linearly in run-time with the size of the dataset used to infer the model.

Lastly, we compared this Markov chain-based approach with another approach using *Multiple Objective Meta-heuristic Search*, studied by a peer in parallel research. We found several advantages and disadvantages for each approach and discovered that the choice for an algorithm depends greatly on the user's requirements.

6 THREATS TO VALIDITY

This section discusses the threats to the internal and external validity of this study.

6.1 Threats to internal validity

The most relevant threat to the internal validity of this study is the sample size. We prepared 5 different datasets of 1000 log traces each, as described in Section 4.2 to mitigate this threat. The total size of all log traces amount to 779 megabytes. We also experimented using several random seeds, as described in Section 4.3, to mitigate the probability of using a biased random seed.

6.2 Threats to external validity

The main threat to external validity is generalizability. This study was a case study on the Consensus Protocol of the XRP Ledger, which might cause one to wonder this study's generalizability. As MASM has only been tested on the Consensus Protocol, there is a probability of the results not being generalizable and thus only applicable to the Consensus Protocol. To mitigate this threat, further research is required using datasets of different systems.

7 RESPONSIBLE RESEARCH

Reproducibility and integrity are the main ethical aspects of this study. This section will discuss how we ensured these aspects were handled ethically during the course of this study.

7.0.1 Reproducibility. To ensure the reproducibility of this study, we made the source code of MASM publicly available in a Github repository [25]. The source code has been thoroughly documented to help one understand it. Furthermore, arbitrarily chosen random seeds were used to ensure the reproducibility of the random numbers. These random seeds are *5, 6,* and *7* for the accuracy experiments. For the run-time experiment, the used random seed is *5*, which is set once at the algorithm's initialisation. Lastly, To increase the reproducibility of this empirical study, all the used log files have been uploaded and are available online [24].

7.0.2 Integrity. During this research, no results were withheld from the presented results in section 5; all collected results are included in the displayed graphs. The results were not filtered nor modified in any way to increase or decrease the projected performance of MASM .

8 CONCLUSIONS AND FUTURE WORK

This preliminary research investigated the effectiveness of using a Markov chain-based algorithm for inferring concise and accurate state models using log analysis. To achieve this, the Markov Algorithm for inferring concise State Models (MASM) was created and evaluated. This algorithm compresses an initial state model, created by parsing logs, using a technique to merge states. MASM implements three different approaches to find states to merge to compress the model, namely: searching for subsequent states with a transition probability of 1, searching for equal rows in the adjacency matrix, and searching for equal columns in the adjacency matrix. When initialized, MASM continues to compress the model until either a maximum result size or minimum accuracy value is reached.

An experimental study was performed on the logs of the XRP Ledger Consensus Protocol. The results show that MASM achieves a lower value for the metrics of specificity, recall, and precision at any rate of compression compared to a random compression algorithm, which randomly merges states to compress the model iteratively. It was also found that MASM shows several knee points at certain rates of compression for the aforementioned metrics, indicating several optimal compression rates for this specific dataset. The most notable knee point for this specific dataset can be found at a compression rate of 52%. Finally, we discovered that this Markov chain-based approach scales linearly in run-time with the size of the dataset, which is promising for MASM 's scalability.

Future research is required to investigate the cause of the high performance of the random compression algorithm; it is hypothesised that this is caused by the manner in which negative traces are generated. Furthermore, integration of an AND condition, rather than an OR condition as described in Section 3.4, on the transitions to states corresponding to several event types remains to be investigated as it might improve the accuracy of the produced models. Finally, measuring the produced model's readability and comparing these results with the readability of models produced by the random compression algorithm also remains to be studied in future work.

REFERENCES

[1] A. W. Biermann and J. A. Feldman. 1972. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Trans. Comput.* C-21, 6 (1972), 592–597. https://doi.org/10.1109/tc.1972.5009015

[2] J. Branke, K. Deb, H. Dierolf, and M. Osswald. 2004. Finding Knees in Multi-objective Optimization. *Lecture Notes in Computer Science* (2004), 722–731. https://doi.org/10.1007/978-3-540-30217-9_73

[3] B. Chase and E. MacBrough. 2018. Analysis of the XRP Ledger Consensus Protocol. (2018). https://arxiv.org/pdf/1802.07242.pdf

[4] G. Fraser and N. Walkinshaw. 2012. Behaviourally adequate software testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation.* IEEE, 300–309.

[5] C. Georgescu, M. Olsthoorn, and A. Panichella. 2021. Modeling System Behaviour from Log Analysis Using Meta-Heuristic Search. (2021).

[6] R.M. Greenwood. 1992. Using CSP and system dynamics as process engineering tools. In *European Workshop on Software Process Technology.* Springer, 138–145.

[7] D. Hand and P. Christen. 2018. A note on using the F-measure for evaluating record linkage algorithms. *Statistics and Computing* 28, 3 (2018), 539–547.

[8] K. Intawong, M. Scuturici, and S. Miguet. 2013. A New Pixel-Based Quality Measure for Segmentation Algorithms Integrating Precision, Recall and Specificity. *Computer Analysis of Images and Patterns Lecture Notes in Computer Science* (2013), 188–195. https://doi.org/10.1007/978-3-642-40261-6_22

[9] Y. Jung. 2018. Multiple predicting K-fold cross-validation for model selection. *Journal of Nonparametric Statistics* 30, 1 (2018), 197–215.

[10] J.J. Li and W.E. Wong. 2002. Automatic test generation from communicating extended finite state machine (CEFSM)-based models. In *Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISIRC 2002.* IEEE, 181–185.

[11] A. Mazak, M. Wimmer, and P. Patsuk-Bösch. 2016. Execution-based model profiling. In *International Symposium on Data-Driven Process Discovery and Analysis.* Springer, 37–52.

[12] S. Messaoudi, A. Panichella, D. Bianculli, L Briand, and R. Sasnauskas. 2018. A search-based approach for accurate identification of log message formats. *Proceedings of the 26th Conference on Program Comprehension* (2018). https://doi.org/10.1145/3196321.3196340

[13] M. Roelvink, M. Olsthoorn, and A. Panichella. 2020. Log inference on the Ripple Protocol: testing the system with an empirical approach. (2020).

[14] A. D. Sahin and Z. Sen. 2001. First-order Markov chain approach to wind speed modelling. *Journal of Wind Engineering and Industrial Aerodynamics* 89, 3-4 (2001), 263–269. https://doi.org/10.1016/s0167-6105(00)00081-7

[15] T. Saito and M. Rehmsmeier. 2015. The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets. *Plos One* 10, 3 (2015). https://doi.org/10.1371/journal.pone.0118432

[16] T. Sen and R. Mall. 2016. Extracting finite state representation of Java programs. *Software & Systems Modeling* 15, 2 (2016), 497–511.

[17] D. Shin, S. Messaoudi, D. Bianculli, A. Panichella, L. Briand, and R. Sasnauskas. 2020. Scalable Inference of System-level Models from Component Logs. *arXiv.org* (Apr 2020). https://arxiv.org/abs/1908.02329v3

[18] M. Sipser. 2006. *Introduction to the theory of computation.* Thomson Course Technology.

[19] W. M. Spears. 1998. A Compression Algorithm for Probability Transition Matrices. *SIAM J. Matrix Anal. Appl.* 20, 1 (1998), 60–77. https://doi.org/10.1137/s0895479897316916

[20] A. Terrasa and G. Bernat. 2003. Extracting temporal properties from real-time systems by automatic tracing analysis. In *International Conference on Real-Time and Embedded Computer Systems and Applications.* Springer, 466–485.

[21] S. Verwer and C.A. Hammerschmidt. 2017. flexfringe: A Passive Automaton Learning Package. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2017). https://doi.org/10.1109/icsme.2017.58

[22] F. Wagner. 2006. *Modeling software with finite state machines: a practical approach.* auerbach Publications.

[23] N. Walkinshaw, R. Taylor, and J. Derrick. 2015. Inferring extended finite state machine models from software executions. *Empirical Software Engineering* 21, 3 (2015), 811–853. https://doi.org/10.1007/s10664-015-9367-7

[24] T. Werthenbach, M. Olsthoorn, and A. Panichella. 2021. Inferring Log-Based Behavioural System Models using Markov Chains. https://doi.org/10.5281/zenodo.5034817

[25] T. Werthenbach, P. Symeonidis, C. Georgescu, and T. Brandirali. 2021. ThomasWerthenbach/ModellingSystemBehaviourMarkovChain: Inferring Log-Based Behavioural System Models using Markov Chains. (Jun 2021). https://doi.org/10.5281/zenodo.5034810

[26] T. Wong. 2015. Performance evaluation of classification algorithms by k-fold and leave-one-out cross validation. *Pattern Recognition* 48, 9 (2015), 2839–2846.

[27] G. Yu, Y. Jin, and M. Olhofer. 2020. Benchmark Problems and Performance Indicators for Search of Knee Points in Multiobjective Optimization. *IEEE Transactions on Cybernetics* 50, 8 (2020), 3531–3544. https://doi.org/10.1109/tcyb.2019.2894664

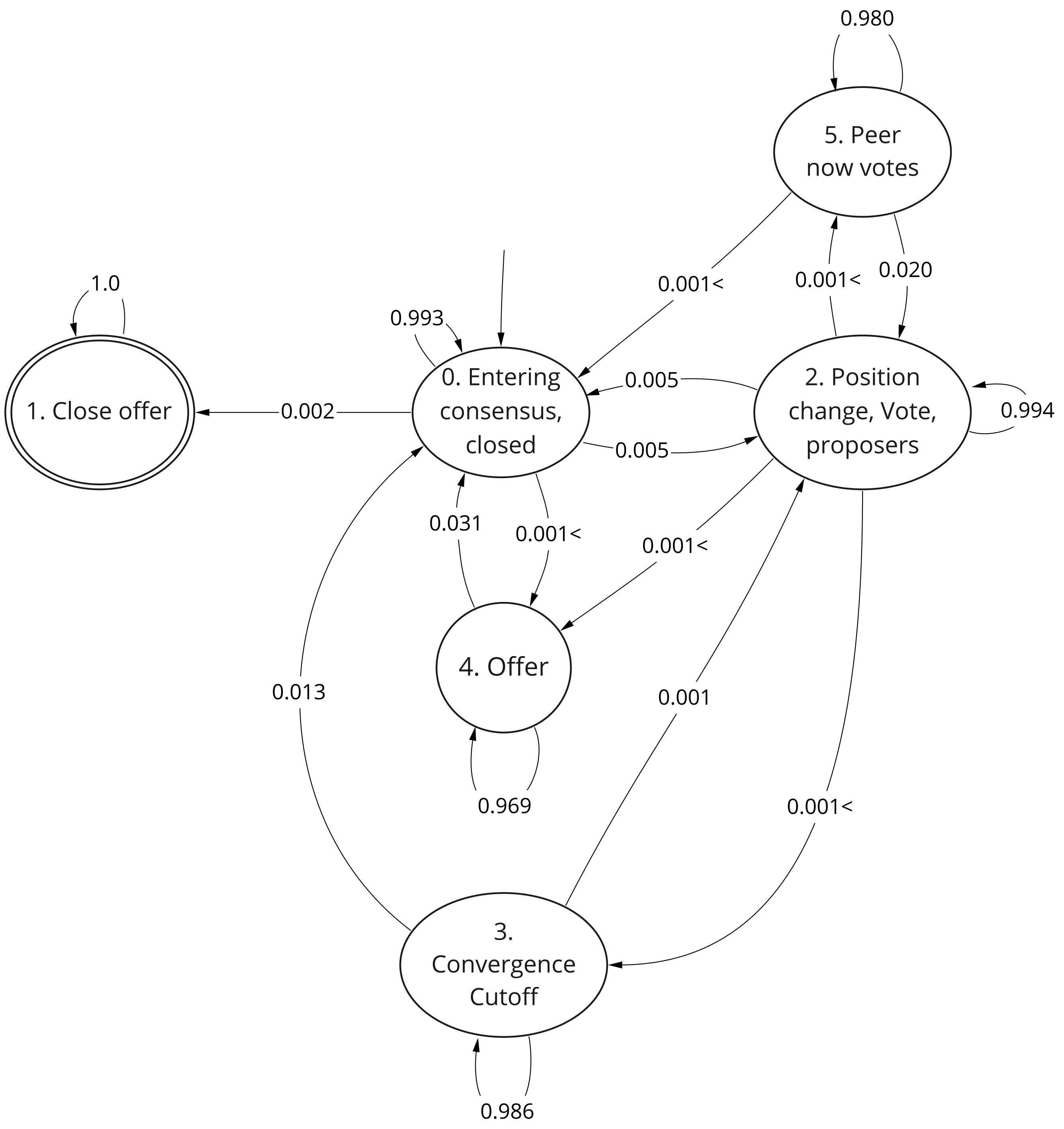
[28] I. Şahin. 2017. Markov chain model for delay distribution in train schedules: Assessing the effectiveness of time allowances. *Journal of Rail Transport Planning & Management* 7, 3 (2017), 101–113. https://doi.org/10.1016/j.jrtpm.2017.08.006

A EXAMPLE LOG STATEMENTS

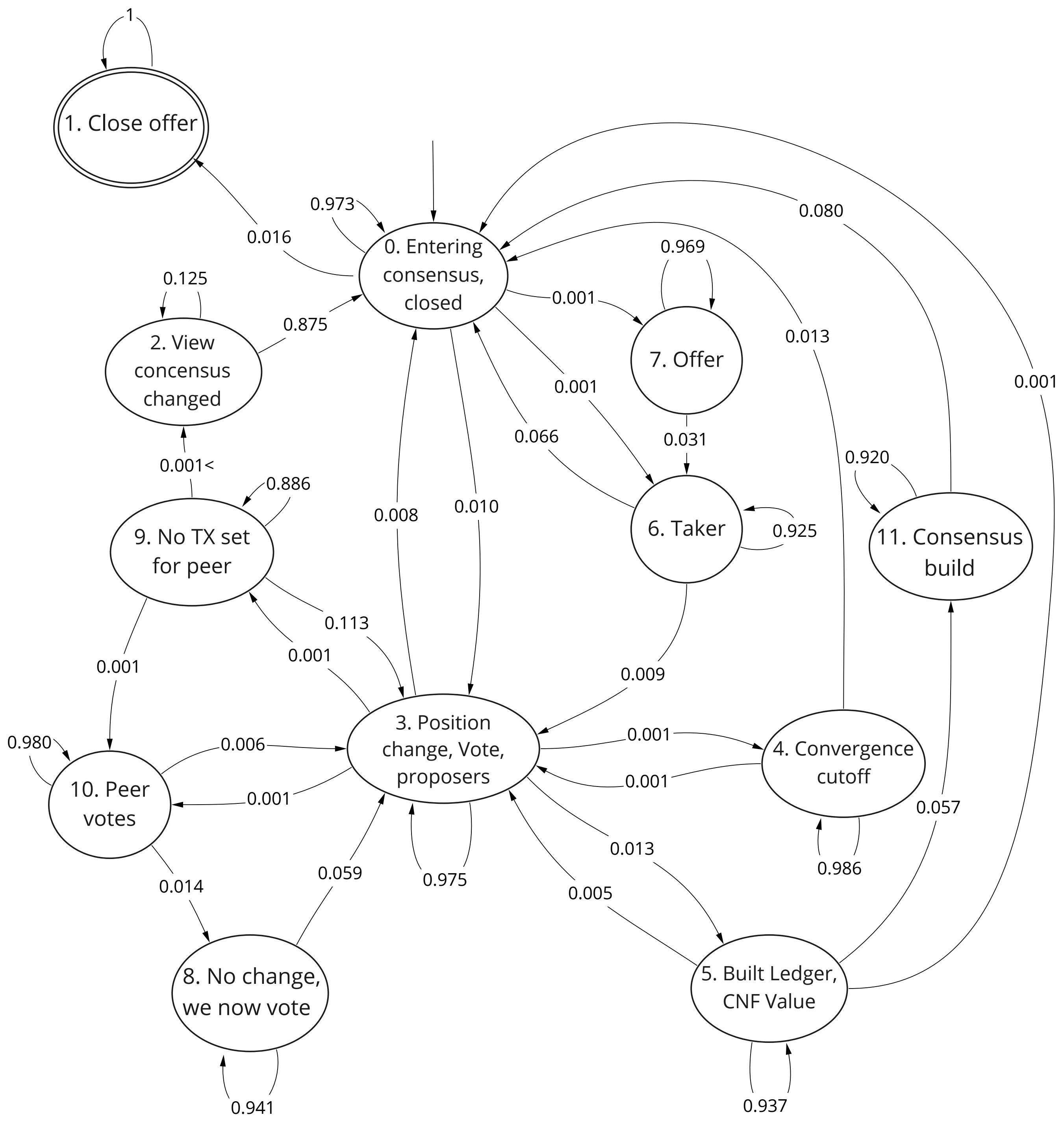
- `2020-Mar-01 18:42:27.620803160 LedgerConsensus: NFO Entering consensus process, validating, synced=yes`
- `2020-Mar-01 18:42:27.621267676 LedgerConsensus: NFO Consensus mode change before=proposing, after=proposing`
- `2020-Mar-01 18:42:27.634688874 LedgerConsensus: DBG Don’t have tx set for peer`
- `2020-Mar-01 18:42:28.494108114 LedgerConsensus: DBG Not relaying disputed tx 5178EEFEEFBF6E7F64432A020AC2374452EB1065A90719B30173A4D13B4B41B2`
- `2020-Mar-01 18:42:28.494154342 LedgerConsensus: DBG Transaction 6E0715D1B20B514014851A5FAA4B9EBD813D48BD259E0802C5A4839DE0D2DBFF is disputed`

B VISUALIZED GRAPHS

B.1 Graph at 95% compression



B.2 Graph at 90% compression


1
1. Close offer
0.973
0.016
0. Entering consensus, closed
0.125
0.875
2. View concensus changed
0.969
0.001
7. Offer
0.080
0.013
0.001
0.001
0.066
0.031
0.001<
0.886
9. No TX set for peer
0.008
0.010
6. Taker
0.925
0.920
11. Consensus build
0.113
0.001
0.001
0.009
0.980
0.006
3. Position change, Vote, proposers
0.001
4. Convergence cutoff
10. Peer votes
0.001
0.001
0.057
0.014
0.059
0.975
0.013
0.005
0.986
8. No change, we now vote
5. Built Ledger, CNF Value
0.941
0.937