Modeling System Behaviour from Log Analysis Using Meta-Heuristic Search

Calin Georgescu
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

Understanding the behaviour of a software system plays a key role in its development and maintenance processes. Unfortunately, accurate and concise models are not always available during development, due to the rapid changes in the structure and scale systems may undergo during this phase. Finite State Machines (FSM) are a natural and prevalent template for modeling system behaviour, and inferring such models through profiling, source code analysis, and log analysis has been an active area of research over the last decades. However, the applicability of existing techniques is restricted by different factors: profiling introduces significant overhead, which is infeasible for many real-time systems; the source code of a system may not always be available for analysis; and current log analysis approaches do not scale well with the number of logs produced by the system. In this paper, four meta-heuristic search approaches are used to create FSM models for the XRP Ledger consensus algorithm from its logs, addressing the scalability problem by approximating a FSM while only exploring a small portion of the search space. Random Selection Stochastic Hill Climber (RSSHC), Tournament Selection Stochastic Hill Climber (TSSHC), Simulated Annealing (SA), and Pareto Simulated Annealing (PSA) are considered. The performance of the model produced by each algorithm is quantified using an accuracy-based evaluation technique, and the accuracy, conciseness, and runtime of the approaches are compared. The results indicate that PSA produces the most accurate and least concise models with a consistent accuracy of over 92%. RSSHC and TSSHC obtain solutions with a slightly lower accuracy, but significantly better conciseness, while SA produces the most compact and least accurate models. All four algorithms produce results within minutes and scale linearly with the size of the problem.

1 INTRODUCTION

Understanding the behaviour of a software system is an essential prerequisite for any developer looking to extend or maintain it. A prevalent modeling technique for software systems is based on Finite State Machines (FSM), a model extensively studied in literature and used for applications such as program analysis and comprehension [11], model checking [9] and test case generation [16, 29]. Unfortunately, due to the fact that constructing a minimal FSM is an NP-Hard problem [4], current techniques for generating accurate state models are computationally demanding and have infeasible run times for large systems. For these reasons, such system models are not commonly used in the practical software development process [34].

Currently, several approaches exist for inferring system models. A common method of model inference is automatic FSM construction through source code analysis [5, 12, 20]. This technique works well for many systems, but suffers from the fundamental limitation that it requires access to the full source code of the target system, possibly including third party dependencies, which are often not available.

An alternative model inference approach is profiling, which consists of measuring the performance of a system's components under different configurations. Several machine learning-based profiling techniques that treat the system both as a white-box [37] and a black-box [17] exist, with different underlying ideas. However, many such approaches underestimate the configuration space of modern software systems, and as a result undersample the number of configurations required to find performance anomalies [18, 37]. Additionally, profiling introduces a significant performance overhead on the target system [32], which is inconvenient for some real-time systems where performance is crucial, like the one considered in this work.

The inference technique used in this work is based on analyzing a system's log traces [39]. This approach presents two key advantages: it requires no access to the source code of the system and it introduces no additional overhead on the execution of the program. Log analysis has been used to infer system-level models [33, 35], with promising results in terms of accuracy, but limited in scalability. In recent years, approaches based on the divide-and-conquer paradigm [26] and parallelization [36] have improved the scalability of log analysis by splitting the task into smaller subproblems. However, a log analysis-based model inference technique that produces an accurate and concise model and which scales well with the number of log traces used as input is not yet available.

In this paper, the scalability problem of current model inference techniques is addressed in a manner which only requires a set of log traces produced by the target system and a configuration of log templates describing what form a particular log entry can take. The goal is to construct a concise and accurate system-level model that describes the behaviour illustrated by the log traces.

To achieve this goal, several meta-heuristic search algorithms are used to minimize an initial (naive) model. First, the naive model is built from a set of log traces and is then minimized through the use of Random Selection Stochastic Hill Climber (RSSHC), Tournament Selection Stochastic Hill Climber (TSSHC), Simulated Annealing (SA) or Pareto Simulated Annealing (PSA). RSSHC and TSSHC are heuristic search algorithms which aim improve a solution by applying small mutations to it and only accept a new potential solution when its evaluation is better than the current one. SA is a probabilistic search algorithm used to find the global minimum of a function, which iteratively improves an initial solution while occasionally allowing so-called *hill climbing* moves (operations which worsen the current solution) in the hope of escaping local minima. SA lends itself to this problem as it has several advantages, namely that its different components can be tuned to accentuate either speed or global convergence, and that it has been extensively studied and has shown very good results in hard combinatorial optimization problems [2, 15, 21, 28]. PSA is a variation of SA that iterates over multiple solutions as opposed to a single one, and seeks to optimize several objective functions. The latter mechanism offers a more accurate way of evaluating a model, as single objective functions may fail to capture subtle trade-offs between the features of a solution.

The evaluation of the prototype implementation of the algorithms focuses on three key aspects: scalability, accuracy and conciseness. Scalability is assessed in terms of the runtime of the algorithm as a function of the number of log traces used to construct the model. To this end, several datasets of an increasing number of traces are used to infer a model, and the average runtime of several executions for each size of dataset is considered. Accuracy is assessed in terms of specificity and recall. Finally, conciseness is measured in terms of the percentage of states merged during the minimization stage. All three evaluations are carried out on datasets composed of log traces generated by the XRP Ledger Consensus Algorithm [8]. This system has been chosen for three primary reasons: (1) it produces large quantities of logs during the many consensus rounds carried out by a blockchain node, (2) the documentation and open source implementation make it possible for an accurate syntax to be described, and (3) it is a prime example of a large scale real-time system for which profiling would have a significant negative impact on performance.

The results indicate that PSA produces the most accurate and least concise models with a consistent accuracy of over 92%, and a size reduced by only 13% on average when compared to the initial model. RSSHC and TSSHC obtain solutions with a slightly lower accuracy, but significantly better conciseness, while SA produces the most compact and least accurate models. PSA is the only algorithm to offer multiple trade-offs between size and accuracy within its solutions. The scalability assessment shows that all four algorithms scale linearly with the amount of traces used in the train and validation phases, and produce models in minutes for a dataset of 1000 log traces.

In summary, the main contributions of this work consist of: (1) a log analysis model inference algorithm based on meta-heuristic search, (2) an open source prototype implementation in Python [6], and (3) the empirical evaluation of the quality of the solution and scalability of the algorithm, tested on log traces produced by the XRP Ledger Consensus Algorithm.

The remainder of this paper is organized as follows: Section 2 contains basic definitions of the terms and concepts used throughout the paper. Section 3 provides a detailed description of the different components of the inference algorithm and reasons behind the choices made. The manner in which the algorithm is evaluated, as well as the exact parameter choices are described in Section 4. The results and their most impactful implications are presented in Section 5. Threats to validity, as well as the reproducibility and the robustness of the results are reflected upon in Sections 6 and 7 respectively. Finally, Section 8 provides recommendations for future work and concludes the paper.

2 BACKGROUND

This section provides basic definitions of the most important terms and concepts used throughout the paper.

Log entry: A log entry is a string of alphanumeric characters created by a software system in order to record some information about the state of the system during execution. Yuan and Zhou [39] identify two components of a log entry: a *static* part that is common for all log entries produced by the occurance of an event, and a *variable* part, which may change based on the circumstances that events occurs in. A simplified version of a typical log entry produced by the XRP Ledger's consensus algorithm has the form: **`2020-Mar-02 DBG: Peer A votes YES on B`**. In this example, the underlined sequences, namely the date, peer names, and the value of the vote are variable parts, while the rest of the entry is static.

Log trace: A log trace is a file containing multiple ordered log entries, usually one on each line of the file. Formally, a log trace is a list of entries $\langle x_1, ..., x_n \rangle, x_i \in \mathcal{E}$, with $\mathcal{E}$ the set of all possible log entries of a system. A trace is called *true* if that trace can be produced by the system during normal execution and *negative* if it cannot.

Log template: A log template is a regular expression that exhaustively models all the forms a particular type of log entry can have. For the previous log entry example, a corresponding regular expression would have the following form: **`[Date] DBG: Peer [A-Z] votes (YES|NO) on [A-Z]`**.

Syntax tree: A syntax tree is a specialized type of prefix tree that efficiently models the set of all possible log entries in $\mathcal{E}$. The *syntax* Σ is the set of all log templates $\mathcal{T}$ that accept exactly all the members of $\mathcal{E}$. Formally, a syntax tree is defined as the tuple $\mathcal{S} = \{\mathcal{R}, \mathcal{N}_s, \mathcal{L}_s\}$, with $\mathcal{R}$ the root, $\mathcal{N}_s$ the set of nodes and $\mathcal{L}_s$ the set of links (or edges). A node n_i contains a *partial template* p_i that matches a part of a log template, while $\langle n_i, n_j \rangle \in \mathcal{L}_s \leftrightarrow \exists e_{i,j}$, an edge between the two nodes. A full log template $\mathcal{T}$ is part of the syntax $\leftrightarrow \exists \langle n_1, ...n_k \rangle$ such that $n_1 = \mathcal{R} \wedge \forall i_{1 \leq i < k}, \langle n_i, n_{i+1} \rangle \in \mathcal{L}_s \wedge \nexists m \rightarrow \langle n_k, n_m \rangle \in \mathcal{L}_s \wedge \mathcal{T} = p_1 p_2 ... p_k$, meaning that a template is part of the syntax if and only if there exists a path in the syntax tree starting from the root and ending in a leaf such that the concatenation of the partial templates of the nodes along that path results in the full template.

Finite State Machine: A variation of the theoretical model of a finite state machine (FSM) put forward by Sipser [27] is used. Here, an FSM is a 5-tuple $(Q, \Sigma, \delta, q_s, F)$, where Q is the set of *states*, Σ is the alphabet or syntax, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, $q_s \in Q$ is the start state, and $F \subseteq Q$ is the set of all final states. Here, states represent the conceptual set of circumstances that the system is experiencing at the time of recording the log statement. The alphabet or syntax describes the possible transitions between states through log templates described by the syntax tree. These transitions are specifically modeled through the transition function, which describes for each state, which neighbouring state can be reached through a particular log template $\mathcal{T} \in \Sigma$. Finally, the start (or initial) state q_s is the state that the machine begins execution in and the set of final states F defines which states result in acceptance at the end of execution. In practice, the FSM is used to decide which traces are produced by the system and which are not. Specifically, an FSM *accepts* a log trace $\mathcal{X} = \langle x_1, ..., x_n \rangle \leftrightarrow \exists \langle q_1, ..., q_n \rangle, q_i \in Q$ such that $q_1 = q_s$ and $q_n \in F$ and for $i = 1, ...n - 1, q_{i+1} \in \delta(q_i, x_i)$.

3 APPROACH

This section provides a detailed analysis of the different stages of the inference algorithm, which can be divided into two large conceptual steps. First, an initial naive FSM model is built from the configuration and log traces given as input, the aim of which is to provide a starting point for the search algorithms. In the second step, RSSHC, TSSHC, SA, or PSA are used to minimize the naive model using a series of state merges, with the goal of balancing a model's accuracy and conciseness. The second step is where the complexity and challenge of the approach lie, particularly in choosing which states to merge, the strategy to use to perform the merge, and the appropriate parameters which best fit the problem. Subsection 3.1 contains a detailed description of the initial model inference. RSSHC and the general framework of the search algorithms are presented in subsection 3.2. TSSHC and the heuristic neighbour selection procedure are highlighted in Subsection 3.3. SA and its components are discussed in Subsection 3.4. Finally, the multiple objective formulation of PSA is outlined in Subsection 3.5.

3.1 Initial model inference

The initial model inference phase is responsible for converting the set log traces provided as input into a FSM yielding a starting point for the minimization algorithms. This section of the program follows a greedy approach, which operates under the constraint described in equation 1.

$$\forall x \in \mathcal{E}, \exists! q \in Q \mid q \text{ describes } \mathcal{T}(x) \tag{1}$$

With $\mathcal{E}$ the set of log entries present in the trace set, Q the set of states in the FSM model, and $\mathcal{T}(x)$ the template describing a log entry x. Informally, this means that to each template describing one or more log entries in the set of traces used to build the initial model corresponds exactly one state in the resulting FSM. This is a corollary of the higher-level assumption that log entries with the same static part are produced by the same underlying software component. The main advantage of this approach is the compact size of the resulting model - which scales up with the minimum number of unique log templates exhaustively describing the log trace set. This benefit is crucial for the scalability of the program, as it allows for the number of traces that the model is built on to vary significantly, while still producing models of comparable sizes: new states are only added to the FSM if log templates that have not been encountered before are observed. The model concisely and effectively describes each sequence of templates that appears in the log traces. The initial model inference procedure is described in Algorithm 1.

An alternative initial inference algorithm was considered without this constraint, which results in the initial model being constructed as a prefix tree. However, under this alternative approach, the number of states in the FSM would scale linearly with the total number of log entries in the log trace set. Empirically, this has been observed to increase the number of states in the initial model by three orders of magnitude for datasets containing hundreds of log traces. Although this alternative approach has an important advantage in that the resulting FSM displays perfect specificity, its size and the complexity required to remove non-determinism in the minimization stage make it an inferior choice in terms of scalability.

Algorithm 1: Initial Model Inference

```
Data: Log trace directory t, Syntax $\Sigma$
initialize model $\mathcal{P}$ with start state $s_0$ and final state $s_f$;
for trace $t_i$ do
    $s \leftarrow s_0$;
    for entry $x_j$ in $t_i$ do
        $\mathcal{T} \leftarrow template(\Sigma, x_j)$;
        if $\mathcal{T} \notin stateTemplates(\mathcal{P})$ then
            $n \leftarrow newState(\mathcal{P}, \mathcal{T})$;
            $addEdge(\mathcal{P}, s, n)$;
            $s \leftarrow n$;
        else
            $n \leftarrow getState(\mathcal{P}, \mathcal{T})$;
            $addEdge(\mathcal{P}, s, n)$;
            $s \leftarrow n$;
    $addEdge(\mathcal{P}, s, s_f)$;
return $\mathcal{P}$;
```

As a result, this approach has only been used to synthesize negative traces for the validation and test sets and is not part of the inference algorithm. Further details on how negative traces are produced are discussed in Section 4.4.2.

3.2 Random Selection Stochastic Hill Climber (RSSHC)

The RSSHC algorithm is conceptually the most basic candidate and is used as a base for the other three, more complex approaches. RSSHC aims to minimize the objective function f, defined in Equation 2.

$$f(x) = \frac{SP + REC + SZ}{3} \tag{2}$$

With *REC* the recall, *SP* the specificity, and *SZ* the relative size of the solution when compared to the initial model. To retain the standard minimizing formulation used in literature, all three metrics are inverted, i.e., defined on the domain $[0, 1]$, where 0 is the global optimum and the quality of the solution worsens as it approaches 1. More details on the procedure used to calculate those metrics are provided in Section 4.4. To minimize this objective function, RSSHC randomly selects two states to merge, generating a neighbour of the current solution in the search space. This procedure is repeated iteratively, replacing the current solution with the generated neighbour only if the neighbour's evaluation is better than of the current model.

The neighbourhood structure employed is generated through *two-state merges* - that is, merging any two states s_1 and $s_2 \in Q_\omega$, except the initial and the final states. Two types of merges are utilized: *OR* merges, which make the FSM more general, and *AND* merges, which make the FSM more specific. Formally, if prior to the merge $\delta(s_0, \mathcal{T}_1) = s_1$ and $\delta(s3, \mathcal{T}_2) = s_2$, then after an OR merge $\delta(s \in \{s_0, s_3\}, \mathcal{T}_1 \vee \mathcal{T}_2) = s_1$, and after an AND merge $\delta(s \in \{s_0, s_3\}, \langle \mathcal{T}_1, \mathcal{T}_2 \rangle) = s_1$ - that is, the sequence of templates $\langle \mathcal{T}_1, \mathcal{T}_2 \rangle$ has to appear in that order for the transition to be possible. The OR merge's tendency to generalize the FSM comes from the self-loops that may occur in the model as a result of consecutive identical templates appearing in log traces. It is important to note that merging two states cannot result in a non-deterministic FSM, since all states are distinct prior to the merge and the merge creates no duplicates.

Algorithm 2: The Hill Climber Algorithm

```
Data: Maximum number of iterations $k_{max}$
generate $\omega \in \Omega, k = 0, T = t_0$;
while $k < k_{max}$ do
    $\omega' \leftarrow selectNeighbour(N(\omega))$;
    $\Delta \leftarrow f(\omega') - f(\omega)$;
    if $\Delta \leq 0$ then
        $\omega \leftarrow \omega'$
    $k \leftarrow k + 1$;
return $\omega$;
```

3.3 Tournament Selection Stochastic Hill Climber (TSSHC)

TSSHC expands on the basis provided by RSSHC by introducing a more sophisticated heuristic in the neighbour selection procedure. Specifically, TSSHC only considers *parent-child* merges - that is, a state $s2$ can only be merged into a different state $s1$ if there exists a transition $\delta(s1, \mathcal{T}) = s_2$. Additionally, a tournament selection algorithm is used to decide which states to merge out of a sample of candidate state pairs.

The rationale behind only considering parent-child merges is that log entries reflect internal changes of the system during execution, either in its state or in the active software component. From this expectation, it follows that fundamentally similar or strongly connected software components often produce adjacent log entries, and merging would be a reasonable improvement to the conciseness of the model. Conversely, merging states that are far apart in the model may result in disconnected software components being stitched together in the resulting FSM. This heuristic is used in TSSHC, SA and PSA.

Selection methods have been extensively studied as they are crucial for many genetic algorithms, where they determine the rate of progress. However, the focus of standard methods such as fitness proportionate selection (FPS) or tournament selection (TS) is choosing parents out of a population for producing multiple offspring [19], which is not the aim of the neighbour selection procedure in the problem formulation at hand. More advanced selection methods have also been recently developed [14, 38] and show promising results, but have not been tested on a similar problem.

In this paper, a tournament selection method is used, as it introduces very little additional overhead to the algorithm, and can even be parallelized, while still allowing the local search to be guided. The neighbour fitness function is defined in Equation 3.

$$\arg\min_{x}\{\forall\mathcal{T}, |\delta(x, \mathcal{T})|\} \tag{3}$$

That is, the fitness function seeks to minimize the number of outgoing edges of the candidate state. The fitness function is applied once in selecting a state, while the state's neighbour is selected randomly. The reason for such a heuristic is that states with fewer outgoing edges are more likely to correspond to weakly connected software components in the original system and as such make for better candidates for being merged with neighbouring states. To decrease the runtime of the algorithm, only a statistically significant random sample of size π is considered. For flexibility purposes, π can be provided as a parameter.

Algorithm 3: Simulated Annealing

```
Data: Cooling schedule t, repetition schedule m, Evaluation function $f$, number of
      iterations $k_{max}$, history array size $r$
generate $\omega \in \Omega, k = 0, T = t_0$;
while $k < k_{max}$ do
    while $m < m_k$ do
        $\omega' \leftarrow selectNeighbour(N(\omega))$;
        $\Delta \leftarrow f(\omega') - f(\omega)$;
        if $\Delta \leq 0$ then
            $\omega \leftarrow \omega'$
        else
            $\omega \leftarrow \omega'$ with probability $e^{-\frac{\Delta}{t_k} \cdot \frac{1}{\Delta r}}$
        $m \leftarrow m + 1$;
    $k \leftarrow k + 1$;
return $\omega$;
```

3.4 Simulated Annealing (SA)

Simulated Annealing (SA) is a convergent local search algorithm that makes use of probabilistic worsening moves (mutations which result in worse evaluations) to escape local optima in the hope of finding a global optima for discrete and continuous optimization problems. The defining aspect of SA is its variable temperature parameter that is lowered over time, which is used to control the probability of accepting a worse solution. This mechanism makes it possible for the algorithm to escape so-called *valleys*, regions of the search space in which no adjacent solution offers an improvement with respect to the evaluation function, with a higher probability earlier in the algorithm and a progressively lower one throughout the execution. In recent decades, SA has been used in a variety of optimization problems, including operations management [21], different scheduling problems [15, 28], and the travelling salesman problem [30], with good results for problem-tailored algorithms. The pseudocode of the SA algorithm is given in Algorithm 3.

The cooling schedule is an integral part of any SA algorithm, which has a vital role in the speed with which the algorithm converges. A cooling schedule is fully defined by (1) an initial temperature t_0, (2) a temperature change rule $u(t)$, (3) the amount of iterations for which a temperature is kept and (4) a stopping criterion. Many schedules have been proposed, tested and surveyed, with temperature update rules ranging from simple geometrically decreasing ones to sophisticated heuristic methods [10, 31]. In this work, the schedule proposed by Lundy and Mees [22] is employed, based on a comparison of five schedules conducted by Cohn and Fielding [10] on several instances of the travelling salesman problem. Based on their comparison, the schedule of Aarts [1], which in some cases outperforms that of Lundy and Mees, has also been considered and could be a good fit for specific problem instances.

Ultimately, the schedule of Lundy and Mees has been chosen because of its faster average convergence and smaller computational overhead. The amount of iterations for which a temperature is kept has been defined as $\frac{1}{10} \cdot max_j(|\mathcal{N}(\omega_j)|)$, a statistically significant sample of the largest neighbourhood in the current front. An initial temperature of 1 has been chosen to allow greater diversification in the initial iterations, and the stopping criterion is left as a parameter to the user. Using these parameters, the cooling schedule is formally defined as the 4-tuple described in Equation 4.

$$(t_0, u(t), m_k, k_{max}) = \begin{cases} t_0 = 1 \\ u(t) = \frac{t}{1+\beta t} \\ m_k = \frac{1}{10} \cdot max_j(|\mathcal{N}(\omega_j)|) \\ k_{max} = n \end{cases} \quad (4)$$

Finally, a modified acceptance criterion has been used in line 9 of the algorithm, where the average of the last r iterations' Δs is used to decrease the acceptance probability, with the aim of restricting the number of accepted worsening solutions. This choice has been made on the basis of the empirical observation that differences caused by mutations tend to be numerically small due to the domain of the evaluation function being $[0, 1]$. For SA, the same neighbour selection procedure is used as in TSSHC.

3.5 Pareto Simulated Annealing (PSA)

First proposed by Serafini [24, 25], Multiple Objective Simulated Annealing (MOSA) has been developed as an extension of the SA algorithm, capable of producing multiple solutions while exploring the search space using a larger number of candidates.

The idea behind MOSA is that instead of providing one solution, a set of non-dominated solutions is generated with respect to multiple evaluation functions. It is said that a solution x dominates a solution $y \leftrightarrow \forall f_i, f_i(x) \leq f_i(y)$ and $\exists j$ s.t. $f_j(x) < f_j(y)$, where f_i are the evaluation criteria. Informally, this means that x is at least as good as y in every criterion, and strictly better in at least one criterion. A *Pareto set* is a set S satisfying the following property: $\forall x \in S, \nexists y \in S$ s.t. y dominates x, meaning that the members of the Pareto set are non-dominated with respect to every other member. Additionally, instead of iterating on a single candidate solution, mutations are independently applied to each model in a set called the frontier. This formulation lends itself perfectly to the goal of FSM minimization, as accuracy and conciseness need to be balanced, and a single solution may not illustrate the possible trade-offs.

Although several variations of MOSA exist, here the adaptable framework proposed by Czyzaak and Jaszkiewic [13] known as Pareto Simulated Annealing (PSA) is used, because of its combination of single-objective evaluation and adaptable weights, which increase the likelihood of escaping local minima. The key aspect of this algorithm is the usage of random adaptable weights as a means of exploring distant sections of the search space. The general formulation of the procedure is given in Algorithm 4.

Where $normalizedWeights()$ generates and normalizes uniformly distributed weights in $[0, 1]$ such that $\Sigma_j \Lambda^x_j = 1$ and $adaptWeights(\Lambda^x)$ increases the weights assigned to superior traits and decreases the ones related to inferior ones, according to the equation 5, before normalizing again.

$$adaptWeights(\Lambda^x, \alpha) = \begin{cases} \alpha\Lambda^x_j, & \text{if } f_j(x) \geq f_j(z) \\ \frac{\Lambda^x_j}{\alpha}, & \text{if } f_j(x) < f_j(z) \end{cases} \quad (5)$$

Additionally, the acceptance function P is defined by equation 6. This function is a modified version of one of the two rules proposed by the original authors, which results in stricter acceptance criterion for this problem, with $\overline{\Delta f_j}$ the mean of the last r absolute differences $|\Delta f^{x,y}_j|$ for feature j. The division by the mean of the last r feature

Algorithm 4: Pareto Simulated Annealing

```
Data: Cooling schedule t, repetition schedule m, evaluation functions f, max. number of
      iterations $k_{max}$, frontier size $s$, scaling parameter $\alpha$, history array size $r$
1  generate $S = \{\omega_1, ..., \omega_s \in \Omega\}, k = 0, T = t_0$;
2  create the set M of potentially efficient solutions from S;
3  while $k < k_{max}$ do
4      while $m < m_k$ do
5          for $x$ in $S$ do
6              $y \leftarrow selectNeighbour(N(x))$;
7              if $y$ is non-dominated by $x$ then
8                  $update(M, y)$;
9              $z \leftarrow closestNonDominated(S, x)$;
10             if not $z$ and not $\Lambda^x$ then
11                 $\Lambda^x \leftarrow normalizedWeights()$;
12             else
13                 $\Lambda^x \leftarrow adaptWeights(\Lambda^x, \alpha)$;
14             $x \leftarrow y$ with probability $P(x, y, t, \Lambda^x, f)$
15         $m \leftarrow m + 1$;
16     $k \leftarrow k + 1$;
17 return M;
```

Δs and the choice of the minimum value among the j features are done to normalize the value of the acceptance criterion, similarly to SA.

$$P(x, y, t, \Lambda^x, f) = min(1, exp(min_j(\frac{\Lambda_j}{t} \cdot \frac{\Delta f^{x,y}_j}{\overline{\Delta f_j}}))) \quad (6)$$

Finally, two noteworthy aspects of Algorithm 4 are that (1) it does not provide the user with one definite solution - the user would have to choose the model that best suits their needs out of the available solutions and (2) the innermost iteration loop over the frontier S can be almost fully parallelized, which stands to bring a significant boost to the algorithm's performance.

4 EMPIRICAL STUDY

This section outlines the various contributing factors to the experimental results. Subsection 4.1 highlights the main goals of the experiments and the research questions they aim to answer. Details about the prototype and the benchmarks used for evaluation are presented in Subsection 4.2. A list of the parameters used for each algorithm is provided in Subsection 4.3. Finally, the experimental protocol is detailed in Subsection 4.4.

4.1 Experimental goal

The aim of the empirical study is to determine the viability of meta-heuristic based algorithms as a scalable model inference technique which produces accurate and concise results. Both scalability and accuracy are of paramount importance to the viability of the algorithm in practical software development tasks such as program comprehension or test case generation, leading to two core research questions the study aims to answer:

RQ1. *How effective are RSSHC, TSSHC, SA, and PSA at inferring a concise and accurate state model for the XRP Ledger Consensus Algorithm?*

RQ2. *How efficiently do RSSHC, TSSHC, SA, and PSA scale in terms of runtime with regard to the number of traces used as input?*

Table 1: Experimental Parameters

Parameter	Value	Remarks
Tournament size π	5	-
Tournament p	0.75	-
K-fold CV k	5	Standard in literature
Cooling Schedule β	0.2546	Used for TSP in [10]
Neighbourhood sample size	$\frac{\lvert\mathcal{N}(x)\rvert}{10}$	Similar value as [31]
History array size r	100	-
PSA Frontier size s	16	Used in [13]
SA and PSA k_{max}	10	Empirical choice
RSSHC and TSSHC iterations	$\lvert states(init)\rvert$	Empirical choice

4.2 Benchmark and Prototype Details

As a benchmark, log traces produced by nodes of the XRP Ledger system have been chosen. The scope of the study has been limited to the Consensus Algorithm (CA), a round-based algorithm that is repeated numerous times during a node's execution which helps determine the state of the blockchain. The CA logs have been filtered out of the system-wide logs on a per-trace basis. Using these traces, 10 separate datasets of varying sizes have been created by selecting distinct random log traces from a pool of size 1000.

4.3 Parameter settings

For reproducibility purposes, the complete list of parameters used in the algorithms is given in Table 1. The majority of the chosen parameters are either standard values used in literature or suggested values by the original proposers of the algorithm. These choices have been made on the basis of the work of Arcuri and Fraser [3], who found that although far from optimal, default parameter values perform sufficiently well, and that tuning is most effective when it is carried out on a varied number of problem instances. Since such tuning is not feasible within the scope of this research and optimal values are not necessary to assess the viability of the approach, default values have been opted for instead.

4.4 Experimental Protocol

This section describes the evaluation strategies and quantifiers used to assess the final set of FSM models. First, the metrics used to describe accuracy and scalability are discussed. Second, the exact methods used to produce the results are described.

4.4.1 Metrics. To quantify the accuracy of a model, two different metrics are used: specificity, defined as $SP = \frac{|TN|}{|TN|+|FP|}$ and recall, defined as $REC = \frac{|TP|}{|TP|+|FN|}$, where an accepted true trace in the test set is a true positive (TP), and a rejected true trace in the test set is a false negative (FN). The converse holds for true and false negatives as well. These measures have been used in many previous works on model inference [23, 26, 35] and capture the model's ability to reject negative traces and accept true ones, respectively. In addition, a model's size is calculated in relation to the size of the initial model produced by Algorithm 1: $SZ = \frac{|states(initial)|}{|states(x)|}$. These three metrics SP, REC, SZ also constitute the three evaluation functions used in Algorithm 4, as well as the components of the single-objective evaluation function described in Equation 2.

The scalability of the algorithm is assessed in terms of its runtime as a function of the number of log traces used to train the model.

4.4.2 Evaluation strategy. Specificity and recall are evaluated using the standard k-fold cross-validation (KFCV) technique. This method splits the dataset into k roughly equally sized folds, of which k-1 are used as the train set and the remainder is used as the test set. Over k iterations, each fold is used once as the test set and k-1 times as part of the train set. Additionally, the evaluation algorithm also splits the k-1 folds composing the train set into a building set used to construct the initial model and an evaluation set, used to approximate REC and SP during the minimization algorithm. This split is done randomly in the current train set, following a 4:1 ratio of traces.

For RSSHC, TSSHC, and SA, the average result over the k iterations of the KFCV algorithm is shown. For PSA, since the algorithm returns many solutions instead of one, only the most accurate solution in each set is considered (in terms of $\frac{REC+SP}{2}$), and the average of the selected values is presented.

To produce negative traces for the evaluation of specificity, a technique similar to those used in other model inference studies [26, 35] has been employed. For each true trace from either the validation or the test sets, between one and three random mutations are sequentially applied. These mutations include (1) swapping two adjacent *log sections*, (2) swapping two random log sections and (3) deleting a random log section. If after performing these mutations, the resulting trace is accepted by the prefix tree built from all positive traces in the entire dataset, then the procedure is repeated. In this context, a *log section* is a sequence $\langle x_1, ..., x_n \rangle$ of consecutive log entries that can be modeled by the same log template $\langle \mathcal{T}, ..., \mathcal{T} \rangle$.

Scalability is measured by running each algorithm on each available dataset a total of 5 times, following a single 4:1 split (into a train set and a validation set). The reported number is the mean of the 5 runtimes for each dataset.

The prototypes of the RSSHC, TSSHC, SA, and PSA inference algorithms have been implemented in Python 3.9, without parallelization at any stage. All experiments have been carried out on a Manjaro 21.0.5 machine running on an Intel Core i7-8550U CPU at 1.80GHz, with 8GB of RAM.

5 RESULTS

This section lays out an overview of results of the empirical study, as well as their most impactful insights. Subsection 5.1 focuses on accuracy and conciseness, while Subsection 5.2 discusses scalability.

5.1 Accuracy and conciseness

The accuracy (measured in terms of REC and SP) and conciseness (measured in terms of compression), as well as the objective evaluation results are displayed in Table 2. A visualization of the results is provided in Figures 1 and 2. The results indicate that all three metrics of evaluation vary significantly between the algorithms.

Of the candidate algorithms, SA consistently produces the least accurate solutions, while also having the largest variation with regard to the number of traces used (cca. +20 pp. between 250 and 750 traces, respectively). However, it also produces the most concise solutions. When analyzing the individual results produced by SA, a discrepancy between the solutions for many of the KFCV iterations stands out: it is common for SA to produce an extremely concise model with perfect specificity and 0 recall (sometimes a 3 state

Table 2: Comparison of accuracy, conciseness, and evaluation between Random Selection Stochastic Hill Climber (RSSHC), Tournament Selection Stochastic Hill Climber (TSSHC), Simulated Annealing (SA), and Pareto Simulated Annealing (PSA)

	Conciseness= $\frac{\lvert states(FSM)\rvert}{\lvert states(initial)\rvert}$					Accuracy = $\frac{REC+SP}{2}$					Objective Evaluation = $\frac{REC+SP+SZ}{3}$					
Dataset size	RSSHC	TSSHC	SA	PSA$_{acc}$	PSA$_{obj}$	RSSHC	TSSHC	SA	PSA$_{acc}$	PSA$_{obj}$	RSSHC	TSSHC	SA	PSA$_{acc}$	PSA$_{obj}$	PSA set size
100	0.54	0.56	0.72	0.27	0.42	0.90	0.85	0.71	0.90	0.87	0.78	0.75	0.71	0.69	0.72	7.5
150	0.45	0.51	0.71	0.16	0.39	0.92	0.88	0.78	0.92	0.89	0.76	0.76	0.76	0.67	0.72	10.9
200	0.54	0.54	0.72	0.15	0.46	0.89	0.91	0.78	0.90	0.88	0.78	0.79	0.76	0.65	0.74	14.6
250	0.48	0.5	0.75	0.18	0.43	0.93	0.92	0.66	0.93	0.91	0.78	0.78	0.69	0.68	0.75	14.3
300	0.49	0.55	0.71	0.12	0.42	0.93	0.91	0.77	0.92	0.9	0.78	0.79	0.75	0.65	0.74	17.4
350	0.48	0.49	0.67	0.15	0.43	0.93	0.92	0.77	0.93	0.9	0.78	0.77	0.74	0.67	0.74	16.6
400	0.47	0.53	0.77	0.08	0.43	0.93	0.9	0.71	0.93	0.9	0.78	0.77	0.73	0.65	0.74	17.9
450	0.48	0.55	0.68	0.2	0.42	0.95	0.92	0.75	0.94	0.91	0.79	0.79	0.73	0.69	0.75	16.7
500	0.51	0.54	0.75	0.14	0.47	0.92	0.92	0.7	0.93	0.89	0.78	0.79	0.72	0.67	0.75	20.5
550	0.51	0.56	0.69	0.1	0.44	0.94	0.91	0.76	0.94	0.89	0.79	0.79	0.74	0.66	0.74	24.9
600	0.46	0.56	0.77	0.12	0.45	0.93	0.92	0.71	0.94	0.9	0.77	0.80	0.73	0.67	0.75	22.5
650	0.48	0.55	0.68	0.11	0.46	0.93	0.92	0.8	0.94	0.91	0.78	0.80	0.76	0.67	0.76	23.6
700	0.48	0.58	0.64	0.16	0.42	0.93	0.92	0.85	0.94	0.91	0.78	0.80	0.78	0.68	0.75	20.5
750	0.46	0.54	0.71	0.1	0.47	0.94	0.92	0.74	0.94	0.92	0.78	0.79	0.73	0.66	0.77	24.5
800	0.52	0.55	0.73	0.11	0.47	0.93	0.92	0.75	0.94	0.89	0.79	0.79	0.74	0.66	0.75	27.9
850	0.5	0.52	0.66	0.12	0.45	0.92	0.92	0.84	0.94	0.9	0.78	0.79	0.78	0.66	0.75	26.8
900	0.51	0.54	0.72	0.09	0.43	0.93	0.92	0.71	0.94	0.9	0.79	0.79	0.71	0.65	0.74	24
950	0.53	0.56	0.73	0.07	0.42	0.93	0.92	0.71	0.94	0.91	0.79	0.80	0.71	0.65	0.75	27.6
1000	0.47	0.53	0.78	0.09	0.46	0.93	0.92	0.7	0.94	0.9	0.77	0.79	0.73	0.65	0.75	27.6

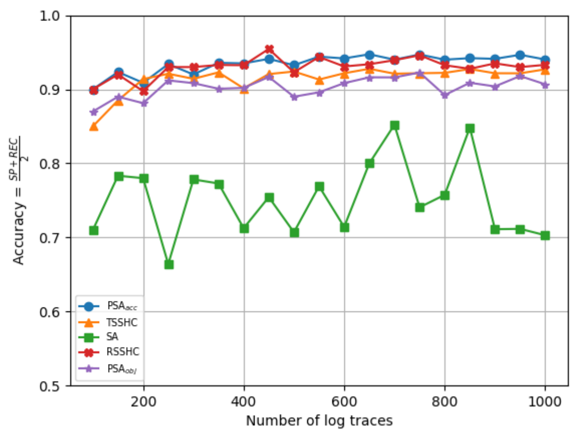

Figure 1: Comparison of the accuracy of the solutions of RSSHC, TSSHC, SA, and PSA, as a function of input size.

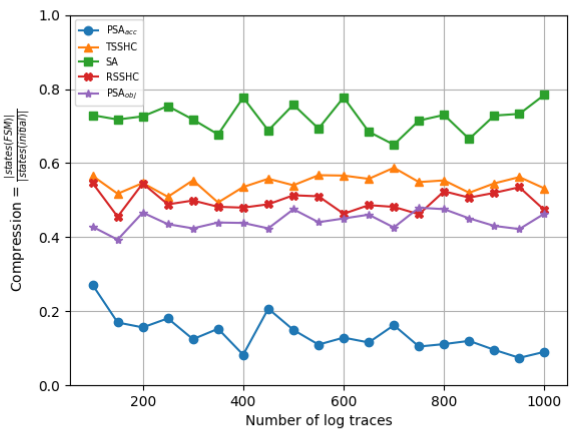

Figure 2: Comparison of the compression of the solutions of RSSHC, TSSHC, SA, and PSA, as a function of input size.

FSM). The reason this occurs is likely the neighbourhood structure: accepting a solution whose evaluation is worse than the previous one may result in a so-called *random restart*, or a jump to an unrelated section of the search space. Similarly, accepting a chain of worse solutions may result in a FSM which due to the chosen neighbourhood structure cannot be improved, since previous merges cannot be undone. The latter situation may cause the occasional 3 state model, which cannot trade off conciseness for accuracy and as such cannot be improved. A more suitable neighbourhood structure may be required to improve the performance of SA.

PSA produces the most accurate results in 16 out of the 18 scenarios. The drastic increase in accuracy (up to 25 pp.) when compared to SA highlights the benefit of the multi-objective approach. In addition to displaying the most accurate results, PSA is the only algorithm that offers the user a choice of trading off accuracy for conciseness: this is highlighted by the PSA_{acc} and PSA_{obj} lines in Figures 1 and 2, which show the most accurate and the best performing (according to the evaluation in Equation 2) models in the Pareto solution set: the user can choose to sacrifice 5 pp. of accuracy for a more concise model, with between 15 and 25 pp. higher compression.

RSSHC and TSSHC perform very similarly and are differentiated by at most 5 pp. both in terms of compression and accuracy. The results help validate the previous conjecture, namely, that the main reason for the comparatively poor performance of SA is the acceptance of worse solutions, as it is the only significant difference

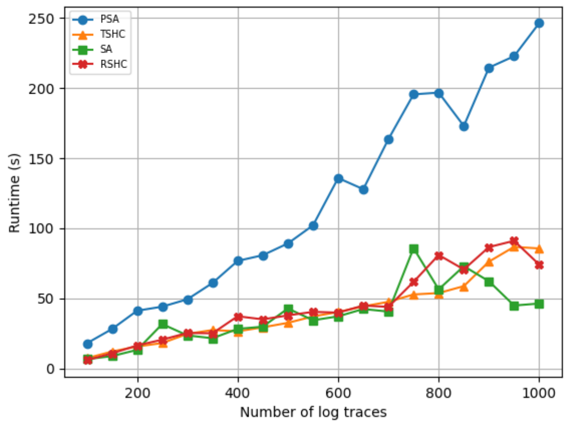

Figure 3: Comparison of the runtimes RSSHC, TSSHC, SA, and PSA as a function of input size.

between TSSHC and SA. Since neither RSSHC nor TSSHC accept solutions which decrease the objective evaluation, the random resets that the SA based algorithms are vulnerable to are avoided. Both RSSHC and TSSHC consistently perform within 5 pp. of the most accurate solution of PSA, while their results are much more concise (up to 45 pp. higher compression). When compared to the best PSA solution according to the evaluation function, both algorithms outperform PSA in terms of accuracy and conciseness consistently. Finally, the similarity of the results indicates that the state selection heuristic stated in Equation 3 does not have a significant impact on the performance of the algorithms. However, tests on a broader range of datasets are required to validate this conjecture. Finally, TSSHC and RSSHC produce the best results in terms of the objective evaluation function described in Equation 2, as they are unaffected by the worsening moves allowed by SA.

Across all algorithms, the number of traces used in the train and validation sets does not have a significant impact on the performance of the algorithm for datasets composed of at least 250 log traces, both in terms of accuracy and conciseness. For fewer traces, a significant decrease in accuracy is apparent in PSA and TSSHC, and to a lesser extent in RSSHC. For PSA, a linear increase in the size of the solution set has been observed.

RQ1: PSA produces the most accurate results, which come at the cost of conciseness. PSA is also the only algorithm to provide multiple possible trade-offs in one iteration. SA consistently gives the most compact and least accurate solutions, but due to random resets, it often produces 3-state models, which are unusable. With over 90% accuracy in the vast majority of runs, RSSHC and TSSHC offer a marginally less accurate and much more compact solution than PSA, but do not allow for a choice to be made by the user. SA and PSA are most likely held back by the neighbourhood structure, which results in occasional random resets.

5.2 Scalability

The scalability results are displayed in Figure 3. The results indicate linear scalability for all candidate algorithms considered.

Of the compared algorithms, PSA is by far the slowest. This is the consequence of evaluating multiple solutions at each iteration, as opposed to a single one. However, PSA has the most to gain from parallelization: each inner iteration could be theoretically sped up by a factor of s, the frontier size parameter. This parameter is also paramount to the performance of the algorithm, as it determines how many evaluation steps are performed for each iteration. The user may determine this paremeter based on their requirements.

RSSHC, TSSHC, and SA all perform similarly and are considerably faster than PSA, with runtimes of around one minute for 1000 traces. The overhead introduced by the tournament selection process in TSSHC and SA is negligeable. Additionally, the runtime of these approaches could be improved by the parallelization of the validation process.

RQ2: The runtime of all four algorithms scales linearly with the number of traces in the train and valdiation sets. PSA is the slowest, while RSSHC, TSSHC, and SA perform very similarly. At under 5 minutes for PSA and around 1 minute for RSSHC, TSSHC, and SA for 1000 traces, all algorithms are viable approaches for the practical software development process from a runtime perspective.

6 THREATS TO VALIDITY

In this section, the possible threats to the validity of the study and the measures taken to ameliorate them are outlined.

Threats to conclusion validity concern the degree to which the conclusions reached about the relationships in the data are reasonable. To enhance the robustness of the results, each considered dataset has been randomly selected from the larger set of 1000 traces, to not introduce any potential bias through a possibly skewed selection procedure. Furthermore, the accuracy tests have been performed using the k-fold cross-validation technique, so as to provide statistical significance to the reported values. However, a broader set of both data points and statistical analysis tests would be required to validate the difference in performance of the considered algorithms and cement the outlined conclusions.

Threats to external validity concern the degree to which the findings of this work may be generalized. The largest threat to external validity is the data that the models have been trained on. In particular, all data in this study has a single system at its origin, which is limiting the scope of the evaluation of the algorithm. To mitigate this, the log traces used to train, validate and test the models have been picked at random from a much larger sample than the study aims to experiment on. In the future, this threat can be tackled by extending the study to include datasets from other systems and compare the algorithms on a per-system basis.

7 RESPONSIBLE RESEARCH

Reproducibility is a key aspect of the scientific method. It provides transparity and is necessary to establish credibility and to allow others to understand one's work. To ensure reproducibility, several measures have been taken. First, the code has been made publicly available through a version-controlled git repository. Each third party component or script has been listed in the repository, along with the specific version that was used to produce the results. The log traces used to train and assess the models have been made publicly available [7]. In addition, the scripts used to produce the reported results as well as the scripts used to create the plots are present in the repository. Lastly, a list of all the used parameters is given in Table 1.

Randomness is at the core of each of the tested approaches, which use it at several points in the algorithm, most notably in choosing a neighbour of the current solution and in deciding whether or not to keep a worse solution in the case of SA and PSA. Randomness is also part of the k-fold cross-validation algorithm, particularly in the splitting of the log trace set into k different disjoint subsets. To reduce the statistical fluctuations caused by randomness in the execution time, the validation procedure scalability has been executed several times, and the average performance has been reported in the results. Additional external factors that could introduce fluctuations have been minimized by performing the tests on the same machine. Finally, the option to provide a seed to the pseudo-random number generator is provided in the code.

8 CONCLUSION AND FUTURE WORK

In this paper, the scalability problem of current log analysis model inference techniques has been tackled using four meta-heuristic search approaches. The proposed inference algorithm first constructs an initial model and then uses a search algorithm to minimize this starting solution. Here, four algorithms were considered: RSSHC, TSSHC, SA, and PSA, which are based on local search and simulated annealing respectively. Each algorithm iterates over either one solution or a set of solutions to which it applies a mutation, seeking to minimize the evaluation function, which is based on the model's size and accuracy.

The empirical evaluation of the algorithm's performance has been conducted on a total of 1000 log traces produced by the XRP Ledger's Consensus Algorithm. The results indicate that PSA produces the most accurate models, followed by RSSHC, TSSHC, and SA. The most accurate solutions of PSA are the least concise, while RSSHC and TSSHC offer a marginally less accurate solution with significantly better conciseness, and SA offers the most concise and least accurate solutions. PSA is the only algorithm to offer multiple trade-offs in the solution set. The scalability assessment indicates that the runtimes of all variants of the approach scale linearly with the number of traces used to train and evaluate the models and that all candidate algorithms are fast enough to be feasibly used in practical software development. PSA is the slowest at cca. 5 minutes for 1000 traces, whereas RSSHC, TSSHC, and SA all perform very similarly at about one minute per 1000 traces.

In the future, a neighbourhood structure which causes fewer random resets for SA based algorithms could be explored, which would likely have the most significant positive impact on the performance of the algorithms. The heuristics that are part of the PSA algorithm could be refined and made more problem specific, especially through parameter tuning. Additionally, the approach would require to be tested on a broader range of systems before any definitive conclusions can be drawn about its general performance. Finally, the integration of the presented approaches with different model inference techniques could also be a valuable contribution.

REFERENCES

[1] Emile Aarts and Jan Korst. 1989. *Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing.* John Wiley & Sons, Inc.

[2] David Abramson, Mohan Krishnamoorthy, Henry Dang, et al. 1999. Simulated annealing cooling schedules for the school timetabling problem. *Asia Pacific Journal of Operational Research* 16 (1999), 1–22.

[3] Andrea Arcuri and Gordon Fraser. 2011. On parameter tuning in search based software engineering. In *International Symposium on Search Based Software Engineering.* Springer, 33–47.

[4] Alan W Biermann and Jerome A Feldman. 1972. On the synthesis of finite-state machines from samples of their behavior. *IEEE transactions on Computers* 100, 6 (1972), 592–597.

[5] Alan W Biermann and Ramachandran Krishnaswamy. 1976. Constructing programs from example computations. *IEEE Transactions on Software Engineering* 3 (1976), 141–153.

[6] Georgescu Calin, Tommaso Brandirali, Pandelis Symeonidis, and Thomas Werthenbach. 2021. *gcalin/WhatTheLog: Modeling System Behaviour from Log Analysis Using Meta-Heuristic Search.* https://doi.org/10.5281/zenodo.5034751

[7] Georgescu Calin, Olsthoorn Mitchell, and Panichella Annibale. 2021. *Modeling System Behaviour from Log Analysis Using Meta-Heuristic Search [Data set].* https://doi.org/10.5281/zenodo.5034777

[8] Brad Chase and Ethan MacBrough. 2018. Analysis of the XRP ledger consensus protocol. *arXiv preprint arXiv:1802.07242* (2018).

[9] Edmund M Clarke, E Allen Emerson, and Joseph Sifakis. 2009. Model checking: algorithmic verification and debugging. *Commun. ACM* 52, 11 (2009), 74–84.

[10] Harry Cohn and Mark Fielding. 1999. Simulated annealing: searching for an optimal temperature schedule. *SIAM Journal on Optimization* 9, 3 (1999), 779–802.

[11] Jonathan E Cook and Alexander L Wolf. 1998. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7, 3 (1998), 215–249.

[12] James C Corbett, Matthew B Dwyer, John Hatcliff, Shawn Laubach, Corina S Pasareanu, Hongjun Zheng, et al. 2000. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium.* IEEE, 439–448.

[13] Piotr Czyżżak and Adrezej Jaszkiewicz. 1998. Pareto simulated annealing—a metaheuristic technique for multiple-objective combinatorial optimization. *Journal of Multi-Criteria Decision Analysis* 7, 1 (1998), 34–47.

[14] Zvi Drezner and Taly Dawn Drezner. 2020. Biologically inspired parent selection in genetic algorithms. *Annals of Operations Research* 287, 1 (2020), 161–183.

[15] Thomas Emden-Weinert and Mark Proksch. 1999. Best practice simulated annealing for the airline crew scheduling problem. *Journal of Heuristics* 5, 4 (1999), 419–436.

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

[17] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wąsowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 301–311.

[18] Xue Han and Tingting Yu. 2016. An empirical study on performance bugs for highly configurable software systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.* 1–10.

[19] Peter JB Hancock. 1994. An empirical comparison of selection methods in evolutionary algorithms. In *AISB workshop on evolutionary computing.* Springer, 80–94.

[20] Bart Jacobs and Alexandra Silva. 2014. Automata learning: A categorical perspective. In *Horizons of the Mind. A Tribute to Prakash Panangaden.* Springer, 384–406.

[21] Christos Koulamas, SR Antony, and R Jaen. 1994. A survey of simulated annealing applications to operations research problems. *Omega* 22, 1 (1994), 41–56.

[22] Miranda Lundy and Alistair Mees. 1986. Convergence of an annealing algorithm. *Mathematical programming* 34, 1 (1986), 111–124.

[23] Leonardo Mariani, Mauro Pezzè, and Mauro Santoro. 2016. Gk-tail+ an efficient approach to learn software models. *IEEE Transactions on Software Engineering* 43, 8 (2016), 715–738.

[24] Paolo Serafini. 1994. Simulated annealing for multi objective optimization problems. In *Multiple criteria decision making.* Springer, 283–292.

[25] Paolo Serafini. 2014. *Mathematics of multi objective optimization.* Vol. 289. Springer.

[26] Donghwan Shin, Salma Messaoudi, Domenico Bianculli, Annibale Panichella, Lionel Briand, and Raimondas Sasnauskas. 2019. Scalable inference of system-level models from component logs. *arXiv preprint arXiv:1908.02329* (2019).

[27] Michael Sipser. 1996. Introduction to the Theory of Computation. *ACM Sigact News* 27, 1 (1996), 27–29.

[28] Kathleen Steinhöfel, A Albrecht, and CK Wong. 2002. The convergence of stochastic algorithms solving flow shop scheduling. *Theoretical computer science* 285, 1 (2002), 101–117.

[29] Ramsay Taylor, Mathew Hall, Kirill Bogdanov, and John Derrick. 2012. Using behaviour inference to optimise regression test sets. In *IFIP International Conference on Testing Software and Systems*. Springer, 184–199.

[30] Peng Tian, Jian Ma, and Dong-Mo Zhang. 1999. Application of the simulated annealing algorithm to the combinatorial optimisation problem with permutation property: An investigation of generation mechanism. *European Journal of Operational Research* 118, 1 (1999), 81–94.

[31] Peter JM Van Laarhoven and Emile HL Aarts. 1987. Simulated annealing. In *Simulated annealing: Theory and applications*. Springer, 7–15.

[32] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-box analysis over machine learning: Modeling performance of configurable systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1072–1084.

[33] Sicco Verwer and Christian A Hammerschmidt. 2017. Flexfringe: a passive automaton learning package. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 638–642.

[34] Neil Walkinshaw, Kirill Bogdanov, Christophe Damas, Bernard Lambeau, and Pierre Dupont. 2010. A framework for the competitive evaluation of model inference techniques. In *Proceedings of the First International Workshop on Model Inference In Testing*. 1–9.

[35] Neil Walkinshaw, Ramsay Taylor, and John Derrick. 2016. Inferring extended finite state machine models from software executions. *Empirical Software Engineering* 21, 3 (2016), 811–853.

[36] Shaowei Wang, David Lo, Lingxiao Jiang, Shahar Maoz, and Aditya Budi. 2015. Scalable parallelization of specification mining using distributed computing. In *The Art and Science of Analyzing Software Data*. Elsevier, 623–648.

[37] Max Weber, Sven Apel, and Norbert Siegmund. 2021. White-Box Performance-Influence models: A profiling and learning approach. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1059–1071.

[38] Fengrui Yu, Xueliang Fu, Honghui Li, and Gaifang Dong. 2016. Improved roulette wheel selection-based genetic algorithm for TSP. In *2016 International conference on network and information systems for computers (ICNISC)*. IEEE, 151–154.

[39] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 102–112.