Mining Specifications using Nested Words

Apurva Narayan
University of Waterloo
Waterloo, ON N2L 3G1
a22naray@uwaterloo.ca

Nirmal Benann
University of Waterloo
Waterloo, ON N2L 3G1
njbenann@uwaterloo.ca

Sebastian Fischmeister
University of Waterloo
Waterloo, ON N2L 3G1
sfischme@uwaterloo.ca

***Abstract*—Parameter mining of traces identifies formal properties that describe a program's dynamic behaviour. These properties are useful for developers to understand programs, to identify defects, and, in general, to reason about them. The dynamic behavior of programs typically follows a distinct pattern of calls and returns. Prior work uses general logic to identify properties from a given set of templates. Consequently, either the properties are inadequate since the logic is not expressive enough, or the approach fails to scale due to the generality of the logic. This paper uses nested words and nested word automata that are especially well suited for describing the dynamic behaviour of a program. Specifically, these nested words can describe pre/post conditions and inter-procedural data-flow and have constant memory requirements.**

We propose a framework for mining properties that are in the form of nested words from system traces. As a part of the framework, we propose a novel scalable algorithm optimized for mining nested words. The framework is evaluated on traces from real world applications.

***Index Terms*—Specification Mining; Nested Word Automaton; Visibly Pushdown Automaton**

I. Introduction

Software performance is highly dependent on the quality of specified specifications. Defining complete and clear specification is a task which can lead to great benefits, whereas incomplete specifications can lead to undetected errors and behavior.

Specifications are given by a set of properties that formally describe the design behavior. These properties may describe temporal behavior, arithmetic relationship between design variables, constraints on inputs and outputs etc. The mined specifications allow us to better understand the system, verify its correctness, and manage possible evolutionary changes.

In this paper, we present a new scalable approach to find recurring patterns in the form of nested words [1] from existing system traces. Given a trace, we match it to a set of parametric pattern templates given in the form of nested words. The matching algorithm is discussed in detail in Subsection IV-B. We also present a ranking module to heuristically rank the most interesting properties found in the system traces. In addition, we present a heuristic that reduces the space and time complexity of the algorithm. Specification mining enables the system designer to automate the formal verification process and provide useful information for fault detection and diagnosis.

With the motivation to address the problem of mining specifications, we propose a technique to mine instances of nested word (NW) templates satisfied by a given system's trace. NWs [1] are a model for representation of data with both *linear ordering* and a *hierarchical nested* matching of items. A NW consists of a sequence of linearly ordered positions, augmented with nesting edges connecting *calls* to *returns*. Although the edges do not cross resulting in a properly nested hierarchical structure, we do allow for some of the edges to be pending. We used nested word automata (NWA) and their finite state acceptors for NWs to check for their occurences in the traces.

It is noteworthy at this point the advantage of using NWA for mining properties in traces. The next promising alternative to NWA for mining nested structures are *Pushdown Automata (PDA)*. The two key advantages of using NWA over PDA are as follows. Firstly, typical implementation of Context Free Languages is using PDAs which are not closed under intersection, complementation, and difference. They cannot decide inclusion and equivalence, and are not determinizable (except Deterministic Context Free Languages). On the other hand NWA overcomes these limitations of PDAs, can have a finite state acceptor representing the regular language of NWs, and exhibit all theoretical properties of Regular Languages![1]. Secondly, PDAs require a *stack* for implementation. Therefore, in mining applications where the inputs to the automatons are either not known or are large, the memory requirement is directly proportional to the length of the input. Whereas, it is constant in the case NWAs which makes them a promising alternative in mining algorithms. NWs with their pre-parsed inputs utilize the best of the two worlds. Motivating the use of NWA for mining specifications can be seen in numerous real-world applications such as:

Detect Code Injection Attacks: Code injection is the exploitation of a computer bug that is caused by processing invalid data. Such injections are used by attackers to introduce the code into a vulnerable computer program and change course of execution. Any such successful code injection in case of nested or recursive calls would violate the nesting structure [9]. Popular case of a code injection are Cross-Site Scripting (XSS) attacks in HTML source [13].

We use a set of simple property patterns for specification in the form of NWs to synthesize a NWA. The NWA is then used as a checker to verify whether traces satisfy the corresponding NW. We provide an algorithm for mining instances of NWs from their templates and also the detailed worst case complexity analysis of the algorithm. The algorithm requires a NW template and system traces as input. The algorithm then uses the distinct events from the traces to replace the event variables in the NW templates with actual events. The resultant *permutations* of the template are NW instances. Intuitively, traces are processed against the NW instances with the help of

the NWA. Further, we define *confidence* and *support* as metrics to evaluate the degree to which a NW instance is satisfied by the traces or, in other words, are interesting and dominant properties [18]. The algorithm reports only the instances which satisfy the given threshold values of confidence and support. Processing all possible instances lead to exponential space and time complexity as shown later. Hence, we utilize the *call* and *return* structure of the NWs and occurrence of pre-post conditions in system traces to heuristically reduce the space and time complexity significantly to obtain the properties.

We also prove that our technique is sound, i.e., a mined specification reported by our algorithms actually satisfies the given thresholds of support and confidence on the provided input traces.

A large segment of software systems, client-server protocols, XML/HTML tags, embedded software systems have nesting structure in their system calls and their responses. We evaluate our technique on real-world datasets that consist of traces produced by applications during various runs on Android operating system, QNX real-time operating system, and HTML sources. Due to constraint on space we present the results on mining of HTML sources. We report the performance of our algorithm on real-life traces in terms of the NW instances reported. We also demonstrate the scalability and efficiency of our approach by running the implementations on synthesized traces of different sizes with different values of parameters such as the number of distinct events, the total number of events in the traces, and the complexity of the NW templates. The key contributions of this paper include:

- An efficient technique for extracting properties in the form of NWs.
- A novel algorithm to mine instances of NWs from given system traces. To our knowledge, this is the first technique for mining specifications using NWAs.
- Computational approaches that optimize the extraction of the specified properties for fragments of NWs.
- An analytic bound on the memory requirements for mining with an empirical validation corroborating the correctness.
- A feasibility and viability study using a HTML source of a commercial website showing the applicability and scalability of the approach.

II. Related Work

Critical and commonly occurring behavioral patterns are typically provided to the mining frameworks, which then mine system specifications of that form. The mining techniques identify a set of specifications that are satisfied by traces w.r.t. certain criteria. Comprehensive review of various mining techniques has been presented in [24].

Many programs lack formal and mined specifications that are valuable as they can be used for a wide variety of activities in the software development life cycle (SDLC). These activities include software testing [6], automated program verification [16], anomaly detection [4], debugging [10], etc. Ammons et. al. [20] argued against the adoption of automated verification techniques are given the complexity in formulating specifications. Different techniques have been developed for mining specifications from templates expressed using regular expressions, LTL [3], timed regular expressions [5], and other custom formats.

A vast majority of tools for mining of properties infer properties in the form of state machines. These tools learn a single complex state machine instantly and extract simpler properties from it. In [22], a model is inferred representing interactions among various components of the software. The tool generates Extended Finite State Machines from the traces of component interactions. They solve the dual purpose of modeling both data and control aspects that are useful for analysis and system verification.

These approaches suffer from two main drawbacks. First, mining of a single state machine from system traces is a NP-hard problem [12] . Second, extraction of formula-based properties from complex state machines still exists.

In another work [8], the authors defined a set of temporal property patterns based on case study of hundreds of real property specification. The main idea behind the exercise was to help designers unfamiliar with formal specifications and static verification approaches. Another work based on the intuition that frequently occurring behavior matches temporal patterns that are likely to be true, is the foundation of Peracotta [20]

The work by Lo et. al. [21] presents a novel way of mining temporal rules based on past-time LTL. They show that past-time LTL can express temporal properties more succintly than future-time LTL. In [11], authors learn a Moore machine from a given set of input-output traces. The work is useful for learning system model for legacy systems. To enhance the capability of specification mining algorithm, authors in [17] combined them with invariant mining to improve the quality of mined specifications. Further strengthening the work of specification mining combined with invariant mining authors in [2] introduce an invariant-counter example guided search for inferring system model. In [23], the authors present a promising approach using guarded finite state machines. They address an important challenge of scalability and specificity of specification mining techniques. Unfortunately, all the techniques mentioned above are unable to handle complex program behavior involving nested architectures. Our work tries to address the problem of mining specifications that are more expressive than regular languages by using NWAs.

Some work has been done in mining correctness of recursive programs using an interpolant based software model [14]. Their method avoids the expensive construction of an abstract transformer and use a NWA of the interpolants. Similar approach using NWA has been used to develop monitors for synthesizing safety properties [25].

III. Background

Generally, regular expressions offer a declarative way to express the patterns for any system property or specification.

Various system properties and specification such as stack-inspection, pre-post conditions, and inter-procedural data-flow are non-regular and hence inexpressible using the classical specification languages based on temporal logics, automata, and fixed point calculi. Not only that, most modern software engineering design methods lead to programs that formulate nested words rather than explicit *GOTO* statement [7]. Therefore, nested words [1] are a promising alternative. They capture both the linear sequencing of positions and a hierarchically nested matching of positions. We can generate a finite-state automata acceptor for nested words [1].

A nested-word automaton is similar to a classical finite-state word automaton, and reads the input from left to right according to the linear sequence. However, at a position with two predecessors, one due to linear sequencing and one due to a hierarchical nesting edge, the next state depends on states of the run at both these predecessors. The resulting class of *regular* languages of NWs has all appealing theoretical properties that the class of classical regular word languages enjoys. The class is closed under operations such as union, intersection, complementation, concatenation, and Kleene-*. Most of the decision problems such as membership, emptiness, language inclusion, and language equivalence are all decidable [1].

In Regular Languages, a sequence of events specifies the ordering of events but not nesting structure of the calls and returns or interprocedural data flow. An abstraction of this sort has been found useful for analysis of certain real time safety critical applications, security of various web applications etc. For example, we might want to modify a formal specification *"a is followed by b"* to a more precise specification with call and return, ***"call** a is followed by an **internal symbol** c and **return** b"*. We use the formalism of NWs for our purpose as it allows for defining explicit calls, returns and internal symbols in the model. We will formally describe our nomenclature.

A. Nested Words

A nested word alphabet is a tuple $\widetilde{\Sigma} = \langle \Sigma_c, \Sigma_r, \Sigma_i \rangle$ that comprises three disjoint finite alphabets–Σ_c is a finite set of *calls*, Σ_r is a finite set of *returns* and Σ_i is a finite set of *internal actions*.

We define nested word automata over $\widetilde{\Sigma}$. The nested word automaton is restricted such that it pushes onto the stack only when it reads a call, it pops the stack only at return, and does not use the stack when it reads internal actions. The input hence controls the operation permissible on the stack.

Definition 1 (Trace and event): The alphabet is a set of distinct events. A *trace* is a set of events ordered by their time of occurrence.

The sequences of events in the trace are ordered by time stamps. The alphabet of events is defined by the system generating the traces. The events have associated meaning pertaining to the functionality of the system.

Definition 2 (Nested Words): A nested word *n* over $\widetilde{\Sigma}$ is a pair $(a_1, \cdot, a_l, \rightsquigarrow)$, where $a_i \in \widetilde{\Sigma}$ and $\rightsquigarrow$ is a matching relation of length l where $\widetilde{\Sigma} = \langle \Sigma_c, \Sigma_r, \Sigma_i \rangle$.

A relation $\rightsquigarrow \subset -\infty, 1, 2, \ldots, l \times 1, 2, \ldots, l, \infty$ of length $l \geq 0$ is a matching relation if the following holds:

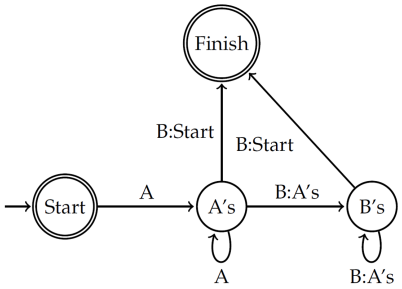

Fig. 1: Finite State Acceptor for NW $\{a^n b^n \mid n \geq 0\}$

- if $i \rightsquigarrow j$, then $i < j$
- if $i_1 \rightsquigarrow j$ and $i_2 \rightsquigarrow j$, then $i_1 = i_2$
  if $i \rightsquigarrow j_1$ and $i \rightsquigarrow j_2$, then $j_1 = j_2$
- if $i_1 \rightsquigarrow j_1$ and $i_2 \rightsquigarrow j_2$, then we have NOT $i_1 < i_2 < j_1 < j_2$

If $i \rightsquigarrow j$, i is a call position and j is a return position. All the rest is an internal position. If $i \neq -\infty$ and $j \neq \infty$, they are well-matched, otherwise pending. Here $e \in \rightsquigarrow$ is a nesting edge.

The novel features here with respect to regular words are the meaning of the atom a_i which represents an event that can be a call, return, or an internal symbol. It is important to note that we use NWs, as defined above, to provide specification templates.

Example: An example of a NW is i**c**iciir**r**i. Here i is an internal, **c** is the first call, c is the second call followed by r, return to the second call and finally **r**, return to the first call. Here it can been seen that nesting structure is preserved.

Definition 3 (NW Templates): A NW template is a NW in which all of the atomic propositions are either event variables or events representing a call, return, or an internal symbol.

A NW template is a template for the specifications that we want to mine. We use the term *event variable* to denote a place holder for an event. For example, the NW template $(\langle a^n . \rangle b^n)$ represents *"a is a recursive call which occurs n times followed by n recursive returns b"*, where a and b are *event variables for calls and returns respectively*. We use p to denote the number of event variables present in a NW template, p_c, p_i, and p_r to represent the number of atomic propositions for calls, internals, and return respectively and p=p_c + p_i + p_r. In the given example p is 2, where p_c=1, p_r=1, and p_i=0. The symbol $\langle$ represents a call and $\rangle$ represents a return.

Definition 4 (NW Instance): Let Π be a NW template. Then, π is a NW instance of Π if π has a NW similar to Π in structure where all the atomic propositions are events (calls, returns, or internals).

Definition 5 (Binding): Let $\widetilde{\Sigma}$ be an alphabet of calls, returns and internal events and let V be a finite set of event variables. Then, a binding is a function $b \colon V \rightarrow \widetilde{\Sigma}$

A NW instance corresponds to a NW template and has an identical NW structure. Applying a binding to the event variables in a NW template creates a NW instance corresponding to that binding. The *binding* is thus a map used to replace event variables with events from the given alphabet.

Definition 6 (Nested Word Automaton): A nested word automaton (NWA) A over an alphabet $\widetilde{\Sigma}$ is a structure (Q, Q_{in}, δ, Q_f) consisting of a finite set Q of states, a set of initial states $Q_{in} \subseteq Q$, a set of final states $Q_f \subseteq Q$ and a set of transitions $\langle \delta_c, \delta_r, \delta_i \rangle$ where $\delta_c \subseteq Q \times \Sigma \times Q$ is a transition relation for call positions, $\delta_i \subseteq Q \times \Sigma \times Q$ is a transition relation for internal positions, and $\delta_r \subseteq Q \times Q \times \Sigma \times Q$ is a transition relation for return positions.

A run ρ of the automaton A over a nested word NW=$(a_1, ..., a_k, v)$ is a sequence $q_0, ..., q_k$ over Q such that $q_0 \in Q_{in}$, and for each $1 \leq i \leq k$. If i is a call position of v, then $(q_{i-1}, a_i, q_i) \in \delta_c$, if i in an internal position, then $(q_{i-1}, a_i, q_i) \in \delta_i$ and if i is a return position with call-predecessor is j, then $(q_{i-1}, q_{j-1}, a_i, q_i) \in \delta_r$.

The automaton A accepts the NW if it has a run $q_0, ..., q_k$ over the NW such that $q_k \in Q_f$. The language $L(A)$ of a nested word automaton, A is the set of nested words it accepts. A language L of nested words over Σ is regular if there exists a nested word automaton A over Σ such that L=$L(A)$.

An example of a finite state acceptor for the NW: $\{a^n b^n | n \geq 0\}$ is shown in Figure 1. A nested word consists of a sequence of linearly ordered positions, augmented with nesting edges connecting calls to returns. The edges do not cross creating a properly nested hierarchical structure. This nesting structure can be uniquely represented by a sequence specifying the types of positions (calls, returns, and internals). The linear and hierarchical nature of the automaton is presented in [1].

B. Dominant Properties

The NW instances generated by the binding contain every permutation of events in the alphabet within the NW template for internal events, calls, and returns of events. There is thus a total of $\widetilde{\Sigma^p}$ possible NW instances. However, these instances contain both interesting and frequently occurring patterns, as well as those that might have been present just a hand-full of times in the trace. We thus use a ranking component to filter the mined set to contain only the dominant instances.

The idea of mining parameters using NWA is novel and we do not have a set of commonly occurring properties available. We propose a set of NW templates which are of interest in software engineering community for monitoring and analysis. We express that a binding and its corresponding NW instance are interesting if the NW instance holds on each trace, thus 100% valid. We use the concepts of support and confidence from [19].

Definition 7 (Support Potential): The support of a NW instance π on a trace t is the total number of time points of trace t which *could* end up in a error state or falsify π.

Definition 8 (Support): The support of a NW instance π on a trace t is the total number of time points of trace t which did not falsify π, and end up in a accepting state.

Definition 9 (Confidence): The confidence of NW instance π on a trace t is the ratio of trace support to trace support potential.

The ranking component we use is a combination of support and confidence. The effectiveness of selecting a meaningful subset of specifications depends on picking a good set of thresholds. Since the total number of mined NW instances is often very large in complex software systems, we would ideally keep the confidence value at 100%.

We will examine different threshold values for both support and confidence and will evaluate the best thresholds for reducing all feasible properties to just the dominant and interesting properties.

IV. Approach

A. Workflow

Figure 2 provides a high level overview of the technique we propose for mining nested properties from system traces.

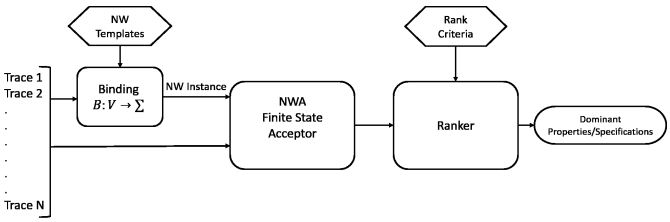

Fig. 2: Property Mining Workflow

The binding function accepts a set of N traces and a NW template. We use execution traces collected during system execution time or source of an application. The event traces are generated using instrumentation already present in the system and may include network traffic logs, operating system logs, program instrumentation logs, etc. The binding function accepts a set of N logs, where $N \geq 1$. From these logs, the function extracts an alphabet $\widetilde{\Sigma}$ of unique events, calls and returns. The NW template is an abstraction of the desired property, a nested relationship of interest for the system. The NW template uses a set V of all event variables (calls, returns, and internal symbols), where the variables range from 0 to p. The binding function binds the set V to the alphabet of events $\widetilde{\Sigma}$ to generate a set of NW instances.

The NWA finite state acceptor evaluates the NW instances on the same set of N traces. The $\widetilde{\Sigma^p}$ NW instances are encoded in the p dimensional incidence matrix that is used by the NWA finite state acceptor to keep track of state, and evaluation of results. As the NWA evaluates each NW event on the trace, it updates the success and failure values in the matrix. When the automaton is finished evaluating the NWs on all the traces, it passes the results from the matrix to the ranker. The ranker uses the results matrix generated by the NWA acceptor to calculate the confidence and support values for each NW instance. The ranking criteria is the threshold values for confidence and support that are used to select only the dominant NW instances.

Let us demonstrate the above workflow through an example. Consider the NW template $((\langle a)^n.c+.(\rangle b)^n)+$, which is a simplified template for the pattern with *n* calls followed by one or more number of internal symbols and then *n* returns that are well matched. The '.' is the concatenation operator and the "+" is the operator for one or more instances of the expression. The template specifies that some call event a occurs *n* times

is followed by another log event c for one or more times, followed by return event b for n times, and this pattern occurs at least once in the execution trace. The property contains three event variables a - call and c - internal symbol, and b - return meaning that the value of p is 3. The binding function will bind the events in $\widetilde{\Sigma}$ to the template and generate an adjacency matrix for the NW instances assuming each event can be a calls, internal, or a return event.

The NWA iterates over the events in a trace and at each new event, evaluates the relevant NW instances. Let us assume the matrix contains an entry where a is bound to an event "send", c is bound to an internal "message", and b is bound to an event "receive". The NWA reads an event "send" in the trace at time 0. According to the property, if the next event in the trace is "receive", the FSM will enter an error state and will increase the failure count for this NW instance to 1 in the matrix. Similarly, if the next event *is not* "receive", but an internal message, the FSM moves to the next state. If the next event is now "receive" then the automaton will enter a final state and will increase the success count for the NW instance.

The number of total NW instances that need to be updated is large and directly proportional to the dimensions of the template, Σ^p. It is also a brute-force approach to evaluate all possible instances where a large chunk of properties turn out to be irrelevant. Most software systems have procedures/-functions that are called, followed by their computations, and then return with an answer. Under normal operation every function call shall have a return associated with it and may have numerous internal operations. This implies that frequency of function calls and return shall be equal. Based on this notion we develop a heuristic to pre-determine the set of calls, returns, and internals for a given trace based on the frequency and desired confidence. All events with equal frequency and in within the desired confidence level define the alphabet for calls (Σ_c) and returns(Σ_r), and the remaining events form the alphabet of internals (Σ_i).

The total number of events in the sets Σ_c, Σ_r, and Σ_i are represented by n_c, n_r, and n_i respectively. If the atomic propositions or placeholders in the NW template for calls, returns, and internals are p_c, p_r, and p_i respectively then, the total permutations that need to be updated is modified to $(\Sigma_c)^{p_c} \cdot (\Sigma_i)^{p_i} \cdot (\Sigma_r)^{p_r}$. This approach takes advantage of the functional paradigm of software design and the architecture of the NWs (with calls, internals, and returns) to mine properties more efficiently with better space and time complexities as shall be discussed later in Section V.

Once the entire trace has been processed, the ranker will iterate over the matrix to calculate the confidence and support values for each NW instance. The ranker will report only the properties that meet the defined thresholds for these metrics.

B. NWA Mining Algorithm

An intuitive way to evaluate a NWA on a system execution trace is to recursively evaluate the NWA according to its semantics at every line of the trace. A high level description of the steps taken by the proposed algorithm are as follows:

- **Representing a Nested Word (NW)** Parse the input Nested Word template and transform it into a finite state acceptor.
- **Representing a trace** Parse the input trace into a linear array representation where each unique event has its corresponding time and event id, *[time, eventid]*.
- **Checking NW instances over traces** Iterate over the trace and process each event by matching them to the relevant NWA instances. Update the *success* and *reset* for the relevant NW instances at each trace event.

Below, in Algorithm 1, is a detailed description of the above steps in form of pseudo code for the case where all possible permutations of NW instances are considered.

Algorithm 1: Nested Word Mining in Traces without heuristic

```
Input:
A trace: TR
Unique events: Σ̃
A Nested Word: NW property pattern with p event variables
Output:
A set of statistically significant properties in the form of NWs
Method: ProcessTrace
Parse NW property pattern to identify number of calls, returns,
  and internals;
Bind each elements of Σ̃ to p as a call, return, and internal to
  generate all possible NW instances;
Create a FSA (Finite State Acceptor) → A for each instance ;
Initialize an incidence matrix of size Σ̃^p instances of NWs ;
Initialize the success and reset counters for each NW instance;
for (each event i in trace TR) do
    for (each NW instance with event i) do
        Update the state s, of FSA;
        Increment successes if the final state s ∈ Q_f;
        Increment resets if the final state s ∉ Q_f;
    end
end
for All the mined properties do
    Evaluate Support - S ;
    Evaluate Confidence - C ;
end
```

In Algorithm 1 and 2, $\widetilde{\Sigma}$ is the events alphabet and p is the number of event variables in the NW template. A denotes the Nested Word Automaton, s denotes the state of the automaton and, S and C denote the *support* and *confidence* metrics of the NW instances on the given trace.

As mentioned earlier that processing of all possible NW instances is computationally expensive and hence, pre-processing of the trace based on a heuristic allows for a faster approach with less irrelevant instances being mined. The heuristic component of the algorithm is presented in Algorithm 2. We only show the modifications to Algorithm 1 as the rest remains the same. The heuristic incorporates the following pre-processing elements as described next. We input an additional confidence parameter θ which is the threshold for choosing the most dominant events with equal frequencies as call/return pairs and the remaining events as internals. The heuristic is applied before the *processTrace* method of Algorithm 1.

Algorithm 2: Heuristic Pseduo-code

Desired Confidence: θ

Heuristic:

1: Calculate the frequency of each unique event;
2: $\theta \times CR$ most frequent events (CR is total events with **equal** frequency) are marked as calls/returns;
3: Remaining events are marked internals;
4: Generate the resultant Σ_c, Σ_i, and Σ_r
5: Bind each elements of $\Sigma_c \rightarrow p_c$ for calls, $\Sigma_r \rightarrow p_r$ for returns, and $\Sigma_i \rightarrow p_i$ for internals to generate all NW instances;

C. Nested Word Patterns

The proposed technique relies on mining patterns that include nesting structure. In order to make use of existing temporal patterns illustrated by Dwyer et. al. [8], we extend them by transforming them into NW Property Patterns. Let us assume we have several call, return and internal events. The NW property patterns are presented in Table I.

Feature	Nested Word
Nested-Call-Response	$([\langle a]^n.[\rangle b]^n);\ n \geq 0$
Nested-Call-Process-Response	$([\langle a]^n.c.[\rangle b]^n);\ n \geq 0$
Nested-Chained-Call-Response	$([\langle a]^n.[\langle b]^m.[\rangle c]^m.[\rangle d]^n);\ n, m \geq 0$
Dual-Nested-Call-Response	$([\langle a]^n.([\langle c]^m.[\rangle d]^n)+.[\rangle b]^m);\ n, m \geq 0$
Nested-Call-Processes-Response	$([\langle a]^n.c + .[\rangle b]^n);\ n \geq 0$

TABLE I: NW Property Patterns

The nested patterns presented in Table I have derived their nomenclature from [8]. The *Nested* refers to a call-return structure in the template as that invokes a function (*call*) which may or may not be followed by an internal *Process*. Finally, the outcome of the function invocation is the *Response* or return.

V. Discussion

We present the following important characteristics of the proposed NW property mining technique: soundness, optimality (in terms of space and execution time), and scalability.

We argue that the technique we proposed and described in this paper is sound. By sound we mean that a mined specification reported by our algorithms actually satisfies the given thresholds for confidence and support in the provided input traces. This is the case because our algorithm continuously keeps track of the success and reset rates for each NW instance in the results matrix.

The proposed property mining technique requires memory space for the multidimensional results matrix, the NWA, and the trace contents. The storage requirements for the matrix are directly influenced by p and $\widetilde{\Sigma}$. We want to encode the matrix to hold the evaluation results of every NW instance generated by binding the events from $\widetilde{\Sigma}$ to the p event variables in the NW template.

By applying the heuristic algorithm, the exponent term is segregated into three components (calls, returns, and internals) thereby reducing the space requirements. The storage required for the FSM is proportional to the number of its states [15]. If n is the depth of the nesting in the NW, then an equivalent NWA uses at most O(n) space.

The comparison of space and time complexity of the two algorithms is presented in Table II. The proofs and details of the analysis have not been included due to space constraint.

Characteristic	Details
Soundness	Our algorithm continuously keeps track of the success and reset rates for each NW instance in the results matrix.
Complete	Our technique examines every NW instance that can be affected.
Run Time Requirement	Worst Case: $O(p \cdot {}^{(\Sigma-1)}P_{(p-1)} \cdot L)$
Memory Requirement	Without Heuristic: O($\widetilde{\Sigma}^p$), With Heuristic: $O({\Sigma_c}^{p_c} \cdot {\Sigma_i}^{p_i} \cdot {\Sigma_r}^{p_r})$
Scalability	$O(L)$ as $L \gg \Sigma$, $L \gg p$ and Σ^p is small for interesting properties

TABLE II: Characterization of the Algorithms

The storage requirements for PDA based mining with the trace is equivalent to the length of the trace, $O(L)$. Our mining approach examines one event from the trace at a time, without needing to store any past or future events for context. It also does not perform any changes on the contents of the trace. Thus the trace is stored only once and never replicated.

The proposed technique is thus scalable with the growing trace size L and with the growing event alphabet $\widetilde{\Sigma}$. This is important since we have no control over the events and traces obtained from real systems. The technique is sensitive to growing values of p and the depth of the NW n. However, as we already mentioned, in practice this will not be a concern since properties remain relatively simple.

We examine the execution time of the main loop in Algorithm 1 for NW templates and present the worst case analysis.

VI. Evaluation

In this Section, we demonstrate the performance and scalability of our approach using a set of real system traces, source, and a set of synthetic traces. The implementation is a R package where the functions are integrated with C++ functions using 'Rcpp'

We developed NW property pattern variants, such as shown in Table I. By using these NW variants as NW templates, we mined NW instances from real-world system traces: QNX tracelogger traces. We used different thresholds for *support* and *confidence* to uncover interesting NW instances. Further, to demonstrate the performance and scalability of our approach, we synthesized traces for different configurations w.r.t. the length of traces, number of variables, nesting depth in a NW template, and the number of distinct events in traces. For the experiments, we used a machine with single 8-core Intel i7-3820 CPU at 3.60 GHz and 32 Gb of RAM.

A. Inferencing HTML Filters

A central concern for a secure web application is an untrusted user inputs. These lead to cross-site scripting (XSS) attacks. The result of which for example could be an untrusted input displayed on the browser. The objective of HTML filters is to block potentially malicious user HTML code from being executed on the server. For example, a security sensitive application might want to discard all documents containing script nodes which might contain malicious JavaScript code (this is commonly done in HTML sanitization). Since HTML5

allows to define custom tags, the set of possible node names cannot be known a priori. In this particular setting, an HTML schema would not be able to characterize such an HTML filter. Therefore, using a set of HTML code, we can mine the properties depicted by the well-formed HTML document. Such as balanced tags (with any level of nesting), presence script node etc. These properties can be used as HTML filters to ensure secure web-applications. One can mine properties from numerous secure web-applications to design a custom HTML filter.

For the purpose of evaluation, we chose a web service that is quite often assumed to be secure and is also prone to code injection attacks-*www.twitter.com* [26]. Of interest from this page is the evaluation of nested HTML tags with the following templates 1) $([\langle a]^n.[\rangle b]^n)+$ and 2) $([\langle a]^n.c + .[\rangle b]^n)+$. The source file from the twitter news feed contained 8000 events, with 75 distinct events is used.

All HTML tags are balanced and it allows us to use our mining technique to generate filters to prevent XSS attacks. Using a sample source from *www.twitter.com* and the NW template $([\langle a]^n.[\rangle b]^n)+$ the algorithm generates a set of nested properties for the HTML source provided. This pattern allows us to model various situations in the HTML source with no `script` tag in between. Since most of the code is generated dynamically, these patterns should remain the same. For example, in a HTML source from twitter the tag `<span>` and `</span>` which provides a way to add a hook to a part of a text or a part of a document is nested with depth two at one instance and depth one at another instance. This property can act as filter for monitoring the HTML source. Violation of this property raises a flag for a possibility of incorrect code or a XSS attack. The algorithm mines this property with a confidence of 100%. This is just an illustration, but HTML source code can contain multiple levels of hierarchy and mining such nested structures with regular languages is not possible. This illustrates the necessity of a mining algorithm using NWs to obtain similar properties of varying nesting depths from traces for post-mortem analysis and developing monitors for secure web applications.

B. Evaluation using Synthesized Traces

Real system traces are suitable for testing the feasibility and usefulness of our proposed technique. However, in real systems, there is little control over the number of distinct events in the trace and the length of the trace.

We synthesized a set of traces by varying the following three parameters: the distinct number of events (the size of the events alphabet $\widetilde{\Sigma}$), the total number of events (length L of the traces), and the number of variables in the NW template p. We used the NW property pattern $([\langle a]^n.c+.[\rangle b]^n)+$. Below, we tabulate each setup of our mining algorithm along with the associated behaviour as shown in Table III.

We compare the performances of Algorithm 1 and Algorithm 2 where the speed up in computation time and space requirements for both cases is in terms of the number of automatons that need to be updated at each iteration. We present the comparison between Algorithm 1 and Algorithm 2 for the three setups mentioned in Table IV for the NW template $([\langle a]^n.c+.[\rangle b]^n)+$. The trend for both algorithms follows the same pattern as stated in Table III. We compare the timing between the two algorithms at different parameters keeping all other conditions the same.

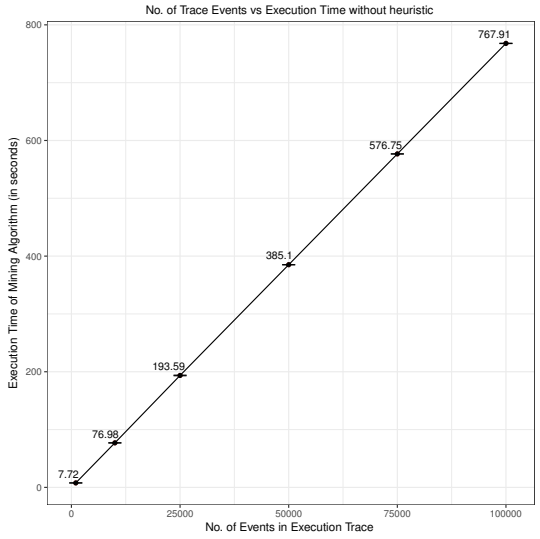

Fig. 3: Varying Total Number of Events without heuristic

Setup	Constant Parameters (Value)	Varying Parameter	Behaviour w.r.t Time
1	$\widetilde{\Sigma}$ (20), p (3)	L	Linear
2	L (10000), p (3)	$\widetilde{\Sigma}$	Exponential
3	$\widetilde{\Sigma}$ (10), L (10000)	p	Exponential

TABLE III: Setup and Results

Conditions	Time w/o Heuristic	Time w/ Heuristic
L=100, 000; p,$\widetilde{\Sigma}$-fixed	767.91sec	9.84sec
$\widetilde{\Sigma}$=40; p, L-fixed	338.61sec	20.69sec
p=3; L,$\widetilde{\Sigma}$-fixed	1.77sec	0.14sec

TABLE IV: Comparison between Algorithm 1 and 2

VII. Conclusions

This paper presented a novel algorithm for mining of NWs in software systems. We present a detailed complexity analysis of the algorithm. We presented two sets of experiments to demonstrate that the algorithm is scalable, robust, and sound. First, we presented experimental results on synthetically generated traces to analyze the scalability of the algorithms with varying total number of unique events, number of variables in the NW template, and varying total number of events in the system trace. Secondly, we validate our framework on industrial strength web applications and mobile framework.

The experimental results confirm that the asymptotic analysis of our algorithm's complexity. We believe that our framework is generally applicable in case of structured systems (e.g. html/xml documents, computer programs, client-server protocols, etc.) and is especially useful for constructing more advanced analysis tools that require specification mining. In

the future, we propose to improve the framework to perform better with the addition of timing constraints to the template. This addition will be useful in real-time safety critical systems, service oriented architectures, etc.

References

[1] R. Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3):16:1–16:43, May 2009.

[2] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 468–479, New York, NY, USA, 2014. ACM.

[3] M. Bonato, G. Di Guglielmo, M. Fujita, F. Fummi, and G. Pravadelli. Dynamic Property Mining for Embedded Software. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 187–196. ACM, 2012.

[4] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 1st India software engineering conference*, pages 5–14. ACM, 2008.

[5] G. Cutulenco, Y. Joshi, A. Narayan, and S. Fischmeister. Mining timed regular expressions from system traces. In *Proceedings of the 5th International Workshop on Software Mining*, SoftwareMining 2016, pages 3–10, New York, NY, USA, 2016. ACM.

[6] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 85–96. ACM, 2010.

[7] E. W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, Mar. 1968.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 411–420. IEEE, 1999.

[9] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 15–26, New York, NY, USA, 2008. ACM.

[10] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 15–24. ACM, 2010.

[11] G. Giantamidis and S. Tripakis. *Learning Moore Machines from Input-Output Traces*, pages 291–309. Springer International Publishing, Cham, 2016.

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

[13] S. Gupta and B. B. Gupta. Cross-site scripting (xss) attacks and defense mechanisms: classification and state-of-the-art. *International Journal of System Assurance Engineering and Management*, 8(1):512–530, 2017.

[14] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. *SIGPLAN Not.*, 45(1):471–482, Jan. 2010.

[15] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to Automata Theory, Languages, and Computation.

[16] Z. Kincaid and A. Podelski. Automated Program Verification. In *Language and Automata Theory and Applications: 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings*, volume 8977, page 25. Springer, 2015.

[17] I. Krka, Y. Brun, and N. Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 178–189, New York, NY, USA, 2014. ACM.

[18] T. D. B. Le and D. Lo. Beyond support and confidence: Exploring interestingness measures for rule-based specification mining. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 331–340, March 2015.

[19] C. Lemieux, D. Park, and I. Beschastnikh. General LTL Specification Mining. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 81–92. IEEE, 2015.

[20] D. Lo and S. c. Khoo. Quark: Empirical assessment of automaton-based specification miners. In *2006 13th Working Conference on Reverse Engineering*, pages 51–60, Oct 2006.

[21] D. Lo, S.-C. Khoo, and C. Liu. Mining past-time temporal rules from execution traces. In *Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, WODA '08, pages 50–56, New York, NY, USA, 2008. ACM.

[22] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, pages 501–510. ACM, 2008.

[23] L. Mariani, M. Pezze, and M. Santoro. Gk-tail+ an efficient approach to learn software models. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.

[24] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated api property inference techniques. *IEEE Trans. Softw. Eng.*, 39(5):613–637, May 2013.

[25] G. Roşu, F. Chen, and T. Ball. Runtime verification. chapter Synthesizing Monitors for Safety Properties: This Time with Calls and Returns, pages 51–68. Springer-Verlag, Berlin, Heidelberg, 2008.

[26] Y. Zhang, X. Wang, Q. Luo, and Q. Liu. Cross-site scripting attacks in social network apis. In *Workshop on WEB 2.0 Security and Privacy (W2SP)*, 2013.