JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2008; **00**:1–21 *Prepared using* smrauth.cls *[Version: 2003/05/07 v1.1]*

Research

Improving Dynamic Software Analysis by Applying Grammar Inference Principles

Neil Walkinshaw*, Kirill Bogdanov, Mike Holcombe, Sarah Salahuddin

The Department of Computer Science, Regent Court, 211 Portobello Street, Sheffield, S1 4DP, U.K.

SUMMARY

Grammar inference is a family of machine learning techniques that aim to infer grammars from a sample of sentences in some (unknown) language. Dynamic analysis is a family of techniques in the domain of software engineering that attempts to infer rules that govern the behaviour of software systems from a sample of executions. Despite their disparate domains, both fields have broadly similar aims; they try to infer rules that govern the behaviour of some unknown system from a sample of observations. Deriving general rules about program behaviour from dynamic analysis is difficult because it is virtually impossible to identify and supply a complete sample of necessary program executions. The problems that arise with incomplete input samples have been extensively investigated in the grammar inference community. This has resulted in a number of advances that have produced increasingly sophisticated solutions that are more successful at accurately inferring grammars from (potentially) sparse information about the underlying system. This paper investigates the similarities and shows how many of these advances can be applied with similar effect to dynamic analysis problems by a series of small experiments on random state machines.

KEY WORDS: Reverse engineering, dynamic analysis, grammar inference

1. Introduction

Dynamic software analysis is the process of executing a program (usually several times) and gathering properties that potentially govern its behaviour. It is particularly appealing because of its inherent precision. Results are limited to a very specific set of concrete program executions, and large quantities of dynamically recorded information can be used to infer strong properties about program behaviour.

The narrow focus that makes dynamic analysis so precise can however also be problematic [1]. The resulting program properties can only be regarded as representative of general program behaviour

Contract/grant sponsor: EPSRC Grant EP/C511883/1

Received 10/2007
Revised 5/2008

(regardless of input or environment) if the supplied set of program executions is 'complete'. For this to be the case, the executions would have to provide a total coverage of every feasible program behaviour.

Depending on the complexity of the program, this set of necessary executions can be prohibitively large. If the assumption is made that the developer is not intricately familiar with the program (which is probable in many dynamic analysis application domains such as program comprehension), the requirement to know the input combinations for every necessary execution renders the whole task of obtaining a complete execution set even less realistic.

In practice, developers are often reduced to supplying analysis techniques with incomplete (often random) sets of executions, which fail to exercise key parts of the program. The results of the analysis must therefore be regarded with a large degree of skepticism. Although certain dynamic analysis techniques and tools have been successfully adopted for programming tasks that do not rely on the ability to generalise, there are many other potentially powerful dynamic analysis techniques that have been neglected because of the time and effort required to provide them with the necessary (complete) set of program executions.

This problem is not unique to dynamic analysis. Grammar inference is an example of another field that is subject to a similar weakness. Here the challenge is to identify the grammar of a language by analysing a sample of sentences. The sample can often be sparse, which means that the resulting grammar is inevitably only partial or even wrong. However, a substantial amount of research in grammar inference has focused on addressing this problem, and has produced a number of relatively successful solutions. These include the use of both possible and *im*possible sentences and the use of active, oracle-driven techniques to guide grammar inference by answering simple questions about system behaviour.

The similarity of the problems and limitations faced by dynamic analysis and grammar inference is striking. This paper argues that many of the solutions in grammar inference, which have resulted in techniques that produce reliably accurate approximations of regular grammars, can be applied with a similar effect to improve dynamic analysis techniques. In exploring this, the paper makes the following three principal contributions:

(1) It provides an overview of the grammar inference problem, and highlights the parallels with dynamic analysis.
(2) It shows how many well-established solutions to the grammar inference problem can be used in dynamic analysis, and demonstrates the value of doing so by a series of experiments on random state machines, which are specifically generated to resemble state machines of software systems.
(3) It elaborates on existing work to evaluate the accuracy of state machines by precision and recall by using systematic model-based testing techniques to ensure that the precision and recall values are not biased by random test case selection.

Section 2 describes the problems that are encountered by traditional dynamic analysis techniques. Section 3 provides an introduction to the grammar inference problem. It also presents some of the key theoretical work that has helped to demarcate the limits of traditional inference algorithms and shows some of the more recent solutions that attempt to work around these limits. Section 4 makes explicit the similarities between the two fields. It shows how some of the solutions that have proved successful in the grammar inference field can just as well be applied to dynamic analysis problems. It also adapts and elaborates an existing evaluation method from the field of dynamic analysis to provide

a more authoritative measure of the accuracy of reverse-engineered state machines. Section 5 presents three small experiments that show the effect of adopting particular grammar inference principles on the accuracy of dynamic analysis techniques. Finally, section 6 offers some conclusions, and outlines some future research.

2. Dynamic Analysis and its Problems

Identifying and recording suitable program executions can be difficult and expensive, which can make dynamic analysis results difficult to interpret and apply in practice. The term 'dynamic analysis' has a broad scope. In this paper we use the following (deliberately loose) definition:

Dynamic analysis: A dynamic analysis technique takes recorded information from one or more program executions, and uses that information to infer rules that govern program behaviour.

The recorded information can for example come in the form of program traces, break point information, or print statements. The program behaviour rules can range from detailed models to simple assertions and invariants. These rules can be generated by automated inference engines or they can simply be in the mind of the developer.

Dynamic analysis is inherently precise; a technique is given a collection of actual observations of program behaviour to start with and, provided that these observations are representative, can construct models that are faithful to the program's behaviour. By using information from program executions, the final models can contain information that could not be obtained from examining the system from a static perspective alone.

Static analysis techniques on the other hand (e.g. symbolic execution or call graph analysis) are forced to consider every conceivable program execution. Because the possible ranges for variables and run-time properties of data structures are not always known, they often end up producing conservative results. Although valid for every feasible program execution, these can often also apply to program executions that are infeasible in practice. This can produce results that are too general to be of practical use to the developer.

However, as with static analysis, dynamic analysis techniques also suffer from major weaknesses. The models or rules they produce about program behaviour are *specific* to the set of executions they are given as input. Although partial models are often useful for providing insights into particular aspects of program behaviour, many programmer tasks require a complete model [1]. For a complete model, an analysis must be given a set of program executions that accounts for *every possible behaviour* in the target model. If this requirement is to be fulfilled, the techniques can rapidly become unscalable. These issues are elaborated in the rest of this section.

2.1. Identifying and recording a suitable set of executions

Obtaining the set of program executions that adequately exercise a system is a notoriously difficult problem. Dynamic analysis techniques usually presume the existence of some 'representative' and 'complete' set of program executions. In practice however there are two barriers that make it almost infeasible to collect them in their entirety.

Firstly, depending on the size and complexity of the (target model of) the underlying system, there can be a vast number of executions that are required by the dynamic analysis technique. For a behavioural model of the system, the set of required executions would be equivalent to a complete

functional test set for the whole system. Obtaining such a test set can be extremely expensive in terms of the amount of time required, the effort required to run the test sets and the expense of storing the executions.

If these reservations are put aside, the task becomes relatively straightforward *if* there is an existing, complete and accurate model of the system; established model-based testing techniques can be used to systematically execute the software system [2]. However, the second barrier arises because dynamic (or static) analysis is usually carried out for applications such as program comprehension, precisely because there is no such model. If there is no prior knowledge of system behaviour, the task of accurately identifying and recording the complete set of necessary executions becomes virtually impossible. Without the existence of a model, users are reduced to using other testing techniques, such as selecting random inputs, pair-wise input combinations or using structural coverage criteria. These techniques are used in the hope that the final set of set of executions will at least approximate the complete set.

2.2. Incompleteness and inaccuracy of dynamic analysis results

Given the probability that the initial set of program executions is incomplete, dynamic analysis results have to be interpreted with a degree of skepticism. This is unacceptable if the intended application relies upon a model that is complete and correct, e.g. the model is required for test set generation, or as the basis for program transformations and optimisations. As a result, these application domains are often dependent on less accurate (but at least complete) static analyses.

Applications of dynamic analysis techniques tend to accept that the supplied set of input traces will inevitably be incomplete to an extent. If this assumption is made, it is reasonable to suggest that the technique should simply produce a result that is as accurate as the set of supplied executions is complete. Although useful for providing certain insights into program behaviour, the specificity of the results makes them difficult to use with confidence. The evaluation of dynamic analysis techniques in terms of their accuracy is often similarly unsatisfying, presenting the accuracy only as a relative value to the set of executions that are provided in the first place, and not as an absolute value with respect to the actual software system itself.

2.3. Scalability of trace analyses

Identifying rules or patterns in program traces can be extremely expensive. Traces can contain vast amounts of information, which can render them cumbersome to store and process. If the information required is only simple (e.g. the highest and lowest values of a particular variable), the size of the trace is unproblematic, because the amount of time taken to identify the required information is linear to the size of the trace (every data point in the trace only has to be examined once). However, as soon as an analysis attempts to identify more complex patterns that require the a more expensive analysis process, scalability inevitably becomes a limiting factor.

3. The Regular Grammar Inference Problem and its Solutions

This section introduces the grammar inference problem, provides an overview of some inherent limits on certain traditional approaches, and introduces some of the most successful recent solutions. It does not provide an in-depth overview of the underlying mechanics of grammar inference techniques. The

purpose of this section is to provide a high-level view of some of the key insights that have led to the most substantial advances in the field. For a more comprehensive overview, there are recent authoritative surveys by Angluin and Smith [3] and Parekh and Honavar [4].

3.1. The Grammar Inference Problem

The problem of grammar inference (also known as grammar induction) was investigated and formalised by Gold [5] in 1967. He investigated the nature of the task facing humans trying to learn new languages, with an aim of producing a theoretical framework for the development of automated techniques that could emulate this learning process. He defined the problem as follows:

Grammar inference technique: A grammar inference technique attempts to learn a grammar G that produces a language $L(G)$, given a set of samples S. The set of samples must contain a set of positive samples S^+ that belong to the language and can optionally contain a set of negative samples S^- that do not. So, denoting Σ as the alphabet of L and Σ^* as the set of all finite sequences over Σ, $S = S^+ \cup S^-$ where $S^+ \subseteq L(G)$ and $S^- \subseteq \Sigma^* \setminus L(G)$.

The simplest class of languages that can be inferred are those that are produced by regular grammars. A machine that produces or accepts a regular language does not require a memory [6] (hence it cannot be used to describe more complex language constructs such as palindromes for example). This lack of memory requirements means that regular languages can simply be described by deterministic finite automata, where transitions are labelled with elements of an alphabet and the automaton represents every valid ordering of those words in a sentence. This representation is particularly appealing because [4]: (a) DFAs are easy to understand and (b) there exist several efficient DFA algorithms that are useful for a number of inference techniques (such as minimisation, determining the equivalence of two DFAs, and determining whether the language produced by one DFA is a super set of the language produced by another).

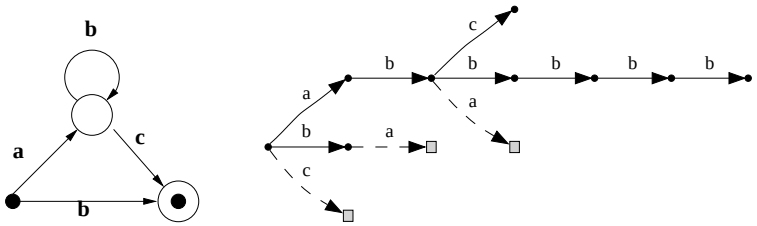
Deterministic finite automaton: A deterministic finite automaton (DFA)is a quintuple $A = (Q, \Sigma, \delta, q_0, T)$, where Q is a finite set of states, Σ is a finite alphabet, δ is the partial transition function $Q \times \Sigma \longrightarrow Q$, $q_0 \in Q$ is the initial state and $T \subseteq Q$ is the set of accepting states. If α is a sequence of symbols in Σ, the target state is denoted as $\delta(q_0, \alpha)$. The string α is accepted if $\delta(q_0, \alpha) \in T$.

Regular grammar inference problem: The regular grammar inference problemcan be defined as identifying the DFA $A(L)$ for some language L such that $\delta(q_0, s) \in T$ for every $s \in S^+$, and $\delta(q_0, s) \notin T$ for every $s \in S^-$.

Figure 1 illustrates the DFA for a simple regular grammar. Positive samples of the grammar correspond to sequences that would be accepted by the machine, and negative samples correspond to strings that would be rejected. As an example, a positive sample could consist of $\{abbbbc, b, abc\}$ and a negative sample could consist of $\{c, aba, ba\}$. Given that we do not have any prior knowledge of the structure of the DFA, a grammar inference technique would attempt to guess it, given only the sets of positive and negative sequences.

3.2. Traditional State Merging Approaches

Most grammar inference solutions revolve around the principle that a state in the DFA of the target grammar is defined by the possible future strings that can stem from it (this notion of equivalence is called the Myhill-Nerode relation [7]). The task of identifying a grammar from S invariably involves



(a) DFA for (ab*c)+b (b) APTA for abc,abbbbb,b (positive) and aba,c,ba (negative)

Figure 1. Examples of DFA and APTA

searching through the strings in S and identifying prefixes in the strings that have identical suffixes. This problem has been shown to be NP-complete in general [8] and was even compared to problems such as breaking the RSA cryptosystem [9].

Nonetheless, since Gold's initial research into the subject, a number of techniques have emerged that can correctly infer a grammar in polynomial time by placing restrictions on certain factors, such as making assumptions about the initial sample of sentences, or adding oracles that can provide additional information to the inference algorithm. Some of those advances have been prompted by a substantial body of theoretical work that establishes the inherent computational limitations on particular solutions to the grammar inference problem. As an example, Gold [8] proved that (for any infinite language) an inference algorithm will require an *infinite* number of positive input sentences to determine the target grammar – thus establishing that any tractable solution would necessarily require some quantity of negative sentences to eventually produce the correct result.

The *state merging* approach was originally developed by Trakhtenbrot and Barzdin [10]. They use S to produce an *augmented prefix tree acceptor (APTA)* - a tree-shaped state machine that represents exactly the samples of positive and negative sequences provided. This is illustrated in Figure 1 (b) (the state numbering can be ignored at this point). The inference process consists of iteratively comparing the suffix-trees of pairs of states, and merging them if their suffixes are identical. As this process continues, the prefix tree acceptor eventually converges upon a minimal DFA representing the target grammar.

Trakhtenbrot and Barzdin's approach requires a *complete set* of samples S (where all possible strings in $S^+$and $S^-$are provided). This of course becomes impossible for any non-finite language, and in practice strings are only provided up to some given length. Although guaranteed to converge upon the correct solution in polynomial time, the approach is cumbersome. It is prevented from making a false merge because this would instantly cause a conflict in the comparison of the suffix-trees. This can however only be achieved when S is complete - as soon as this is not the case, incorrect merges can occur, and it becomes increasingly probable that the technique will ultimately end up producing an incorrect machine that may not even be consistent with the strings in S. Although polynomial, the simplistic strategy of attempting to merge nodes in a breadth-first manner tends to result in a large number of unnecessary state merges.

Subsequent work by Lang [11] and Oncina and Garcia [12] has resulted in the RPNI (Regular Positive Negative Inference) algorithm*. This improves upon the Trakhtenbrot and Barzdin algorithm because it will guarantee a machine that is consistent with S even if S is not complete. Oncina *et al.* [14] also show that the algorithm can still produce an accurate machine, provided that S contains a *characteristic* sample of the target grammar. Informally this means that, given the minimal DFA for some target language, S is characteristic if it contains positive sequences that cover every transition, as well as a sufficient set of positive and negative sequences to differentiate between any pair of non-equivalent states. For a more formal definition the reader is referred to work by Dupont *et al.* [15].

The requirement for a characteristic sample of the target grammar is however still impractical. Depending on the size and complexity of the target machine, the number of required sequences in the characteristic sample can be extremely large. Constructing a characteristic sample requires a substantial amount of prior knowledge about the underlying system, ultimately rendering the requirement unrealistic for a large number of practical applications.

3.3. Active State Merging

Traditional state-merging techniques fail when the supplied sample is sparse†. They use rigid techniques such as breadth-first search to construct work lists of possible state merges. If the provided sample of sentences is sparse, there is not going to be enough information to prevent a wrong merge from happening. Because the selection of merges is in effect arbitrary (not based on any evidence to support the similarity of a pair of states), there is a high probability that an erroneous merge will occur. When an incorrect merge occurs, subsequent merges inevitably compound the error, resulting in a highly inaccurate final machine. Lang [11] reinforces this point by demonstrating empirically that for a sparse sample of sequences, a traditional merging algorithm will only *approximately* identify the correct target machine if the size of the (random) sample is exponential in the size of the target machine.

Recent advances in grammar inference have addressed this problem by (a) augmenting traditional techniques with the ability to question an oracle in an efficient manner and (b) making use of the available data to identify suitable state merges by using heuristics instead of rigid work lists. These two approaches are presented below. The section then concludes by introducing Dupont's QSM technique, which combines the two approaches and will form the basis for the implementation that we use in the experiments in section 5.

3.3.1. Incremental and Active Algorithms for State Merging

One of the major weaknesses of the conventional RPNI and Trakhtenbrot and Barzdin algorithms mentioned above is the fact that they require the entire set of samples S in advance (this is referred to as *presentation from given data*). Listing every necessary sample in S to produce a correct machine can simply become too expensive and often requires so much prior knowledge about the underlying grammar that it practically undermines the purpose of inferring the grammar anyway.

*The equivalence between RPNI and Lang's algorithm is discussed by Garcia *et al.* [13]
†A sample is termed *sparse* if it is neither complete nor characteristic.

Dupont developed the RPNI2 algorithm [16], which allows for *sequential presentation* of data. Instead of requiring every string in S to begin with, positive and negative strings can be fed to the algorithm in an iterative fashion (as they become available). Although the same result could of course be achieved by simply restarting the conventional RPNI algorithm with the extended sample, the RPNI2 algorithm achieves the same effect much more efficiently. Although the experimental results show only a minor improvement in terms of performance, the real benefit lies in the usability, as the scenario of sequential presentation is more likely to occur in practice.

Although the incremental extension renders the algorithm easier to use, it still does not solve the more fundamental problem of identifying the relevant sets of strings that are required to arrive at a correct machine. This would require an *active* approach. This was the subject of subsequent work by Damas *et al.* [17]. They elaborated on Dupont's earlier work by producing an extension of RPNI that is not only incremental, but also active; it does not depend on the user to think of suitable inputs, but asks the user *membership queries*, and the user merely has to answer with a 'yes' (the full query is valid) or identify the point in the query that is invalid. The queries are not posed randomly; due to the fact that a characteristic set of samples is proven to generate a complete and consistent grammar, questioning process aims to 'home in' on this set. Each proposed merge of a pair of states is tested, by producing questions that should highlight whether a pair of states is different. If they are, these new negative sequences can be used to produce a more accurate model, and the merging process is restarted.

This questioning strategy is valuable because it allows the user to start off with a relatively sparse set of initial sentences. The user does not need an extensive prior knowledge of the underlying grammar, and the questioning technique will attempt to ensure that the final model is nonetheless relatively accurate (provided that questions are answered correctly). By attempting to identify the characteristic set, the question generation process ensures that only essential information is gathered from the oracle.

3.3.2. Heuristic State Merging

A major source of inefficiency in the traditional algorithms lies in their rigid approach to selecting candidate pairs of states to merge. As an example, RPNI states are simply merged by ranking states in the APTA (see figure 1(b)) in terms of the length of their prefixes. The ranking is according to standard lexicographical order, and pairs (a, b) are chosen so that the rank of $a \leq b$ (i.e. a is closer to the root of the tree). Ultimately though merges are chosen arbitrarily, in the hope that a pair of states is either equivalent, or that there is otherwise a string in S that will prevent an invalid merge from happening. If a false merge does happen, and there is not enough information in S to prevent it from happening (assuming that no questioning procedure as detailed above has been implemented), the resulting machine will be wrong, and the error will merely be compounded by future merges.

This problem has been addressed by a variety of recent solutions in the grammar inference community, many of which have been spurred by competitions to promote the development of better algorithms. The simplicity of the problem - to learn a deterministic finite automaton, makes the format of such competitions relatively simple as well. All that is required is a random FSM (it must be deterministic and finite), along with a set of strings to train the candidate algorithms, as well as a set of strings to test the accuracy of the resulting model.

The Abbadingo One competition [18] is perhaps the most famous, because it spawned two techniques that have had a major impact on the field: the Blue-Fringe approach for selecting potential pairs of states to merge, along with the EDSM algorithm, which can be used to assign a score

1. Colour all nodes in the APTA white, mark the root node as red
2. **while**(! all nodes are red)
   (a) Mark all nodes that are adjacent to red nodes blue
   (b) **If** a blue node can't be merged with any red node
      i. mark as red
   (c) **else**
      i. Compute compatibility scores for all red-blue pairs
      ii. merge highest-scoring pair

Figure 2. Blue-Fringe EDSM algorithm

to a particular merge pair. These two techniques, especially when used in conjunction, can vastly improve the inference process, and have become a practical base-line approach to inferring grammars in practice.

The Blue-Fringe algorithm can be used to, at any point during the execution of a state-merging algorithm, identify a relatively small set of candidate pairs of states to be merged. The number of candidate pairs that would need to be considered by a conventional merging algorithm at any one point is usually huge; the conventional RPNI algorithm, for example, can in theory try to merge every node in a given sub tree of the APTA with a single node near the root. The Blue-Fringe algorithm significantly reduces this, by partitioning the set of nodes in the APTA into three groups. Red nodes represent a core of nodes that are considered to be mutually unmergable, the 'blue fringe' is a set of nodes that are adjacent to the core of red nodes, which are considered to be potential merge candidates, and the rest of the nodes are white. The algorithm is shown in figure 2, step 2(c) is elaborated below.

Given the set of red-blue node pairs, a conventional algorithm would simply go about attempting to merge them in an arbitrary order. This is however often overly optimistic, and heavily relies on the availability of sufficient samples to prevent a false merge from happening. To lessen this reliance, De la Higuera *et al.* [19] suggest that at any given point an algorithm should order the potential merges in terms of the likelihood of the pair of states being equivalent. However, their proposed solution contained a number of flaws which rendered it relatively ineffective on certain data sets. Price's EDSM algorithm [18] (winner of the Abbadingo competition) adopts De la Higuera *et al.*'s idea, and provides a solution that is more effective over a wide range of sample sets. For a given set of potential pairs of states to be merged, it can assign a compatibility score to each pair. This is computed by comparing the state transitions that happen after each set of nodes, and counting the number of transitions that are equivalent. If a conflict is identified (e.g. a path x leads to a reject state from q, but to an accept state from q'), the pair receives a negative score and is not merged. When used together, the Blue-Fringe and EDSM techniques tend to produce a significant improvement over other state merging techniques.

Dupont's QSM approach [15] is perhaps one of the most usable and effective inference techniques developed so far, because it is an active approach that incorporates the Blue-Fringe and EDSM pair selection strategy. Merges are selected by using the Blue-Fringe strategy and appropriately scored and ordered using the EDSM algorithm. As the merges are processed, the QSM algorithm generates lists

of questions for each merge to (a) check that the merge is valid and (b) garner missing information to prevent false merges from being attempted in the future. Depending on the initial sample S, this can produce a highly accurate grammar, and substantially reduce the amount of time and computation effort required by the inference algorithm.

3.4. Alternative Approaches

It is important to note that the overview presented in this section is only partial. The point is to convey the main attributes of effective grammar inference solutions to the reader, to set the context for the rest of the paper. There are however a number of other approaches (notably Angluin's $L*$ technique [20]), that incorporate negative information and are active. The QSM and $L*$ techniques differ fundamentally because the $L*$ algorithm exhaustively explores the entire state space of the learned machine, whereas the QSM algorithm takes S and generalises only from the presented set of traces. Although $L*$ is always guaranteed to produce a complete and correct machine, it can become prohibitively expensive when a sparse set of samples is provided for a complex state machine, which is why we argue that the (possibly incomplete) QSM technique is more suited for the domain of dynamic analysis.

4. Applying Grammar Inference Principles to Dynamic Analysis

There is an obvious overlap between the problems that are addressed in the grammar inference and dynamic analysis communities. Both attempt to derive facts or models from a finite sample of observations. Regular grammar inference aims solely to infer a DFA, whereas dynamic analysis has a broader range of potential target models. This section points out some of the parallels between the two fields. If the purpose of dynamic analysis is to infer a model that is equivalent to a DFA, the analysis problem can be recast as a grammar inference problem, and the solution can take advantage of the many advances that have made regular grammar inference more tractable. Even if the target of dynamic analysis is not to produce a DFA-equivalent model, there are still many sufficiently general principles that can be applied regardless of the target model.

The next subsection will present an overview of existing approaches to dynamic analysis that are inspired (at least to some extent) by grammar inference approaches. Section 4.2 presents the grammar inference principles that have not yet been applied in the domain of dynamic analysis techniques. Although the primary dynamic analysis application considered here is the use of execution traces to reverse engineer state machines of software systems, the principles are sufficiently general that they can be effectively applied to most other dynamic analysis applications that involve generating a model from a limited set of observations.

4.1. Existing solutions inspired by grammar inference

The analogy between grammar inference and dynamic analysis was first realised over 30 years ago when Biermann and Feldman [21] proposed their $k-tails$ state-merging algorithm that could generate state machines from sample executions. The $k-tails$ algorithm is a variant of Trakhtenbrot and Barzdin's passive state merging algorithm that considers two states equivalent if they are buth succeeded by a string of length k. The best value for k depends primarily on the developer's judgment.

A large k value may make the state comparison process more costly, but will produce a more precise machine, but will often prevent merges that should happen, merely because there is a lack of evidence to justify them. A small k value can result in the merging of non-equivalent states and produce incorrect machines, but requires less computation time.

A number of papers exist that explore the link between grammar inference and dynamic analysis. Most of these augment the $k - tails$ algorithm [22, 23, 24, 25, 26, 27]. However, these techniques restrict themselves to a rigid notion of dynamic analysis that does not permit them to take advantage of some of the substantial advances that have taken place in grammar inference. The conventional dynamic analysis framework is subject to those problems that were introduced in section 2; it assumes the provision of a single 'representative' selection of execution traces (i.e. it is passive) and it conventionally only accepts *positive* traces of program behaviour (no negative information). Gold's work [8] proved that, for any infinite language (which for us translates to any software system with loops - i.e. the majority) a dynamic analysis technique will necessarily require an infinite number of program executions to produce a definitive result. The only alternative is that *every* positive sample up to a given length is provided, which in the case of dynamic analysis becomes impossible for any non-trivial system.

Consequently, the result of a dynamic analysis based on a finite set of program executions is inevitably only an approximation of the target system and is, without any negative information, inherently prone to over-generalisation. In practice this means that the model that is presented by such an analysis will exactly show some set of rules that govern the provided set of program traces, but will not be able to make any useful inferences about the general system behaviour; it cannot infer impossible behaviour and, due to a combination of insufficient negative information and incomplete set of samples, any steps that attempt to infer behavioural rules beyond the supplied set of samples will probably be false.

This problem is addressed (at least to an extent) by reverse engineering approaches that are based on Angluin's $L*$ algorithm [20]. A large number of techniques exist that use the $L*$ algorithm to build models from software systems [28, 29]. These are interesting because they are active, and incorporate negative information about system behaviour to produce a complete model of system behaviour. However, as mentioned in section 3.4, their nature of exhaustively probing system behaviour and generating vast numbers of queries renders them prohibitively expensive for non-trivial systems. These techniques are not intended for what would be considered a typical dynamic analysis scenario (i.e. the developer has a set of traces from some arbitrary system and wants to generate a model from them). These are instead used for relatively specialised software systems, where the execution of thousands of tests is not a concern.

4.2. Applying techniques from grammar inference to dynamic analysis

The problems with dynamic analysis that are mentioned in section 2 are unnecessary. A number of the advances that have vastly improved the performance of the grammar inference algorithms introduced in section 3 can be adapted to dynamic analysis. Before showing how this can be done, we make the following assumptions about the underlying system and the collected program traces:

1. We assume that the model of the underlying system is static and deterministic (i.e. the system is not self-modifying in an way). This means that, for multiple executions, the system must respond in a consistent manner.

2. Traces of program executions retain the sequential order in which program elements are executed. Ultimately a trace should be properly abstracted to represent a path in the target machine - Ammons *et al.* [22] describe how this can be achieved.

Preprocessing traces to facilitate storage and analysis

In grammar inference, state merging approaches store the initial set of sequences in a single structure, called the (augmented) prefix tree acceptor (see section 3.2). Depending on the characteristics of the provided set of sentences, this can substantially reduce the amount of storage space required. Reiss and Renieris [27] make this point with respect to the storage of traces for dynamic analysis.

Systematic merging algorithms such as RPNI take advantage of the structure of the APTA. The compression of the traces into a tree means that there are fewer nodes to compare to each other. This can be further reduced by adopting the Blue-Fringe approach (section 3.3.2), which takes advantage of the tree structure to more efficiently evaluate potential merges in a systematic manner.

It is however important to note that the benefits of storing traces in an APTA range beyond state-merging techniques. Any dynamic analysis technique that needs to compare different trace points can benefit from a compact tree representation. The technique becomes more efficient because the number of comparisons that have to be made are reduced, without losing any essential information about traces themselves. Also, generally, the representation of a set of traces as a tree makes them amenable to well-established tree-based search algorithms, a potentially important step towards improving the scalability of trace analysis algorithms (see section 2.3).

Incorporating negative information

As established by Gold, the use of only positive information (i.e. valid program executions), without any negative input cannot produce a definite result with a finite amount of input. This can only be achieved by including a sufficient amount of information about what *cannot* happen (negative information) to prevent the analysis from over-generalising from the provided set of traces. Whereas traditional grammar inference techniques can usually only rely upon the manual provision of negative strings, software analysis is better-equipped, with a range of (largely automated) techniques that can be used to generate this information in large volumes. Some possible sources are presented below:

- **Static analysis** can provide a substantial amount of program behaviour that is *im*possible by examining the complement of its results. The most suitable static analysis approach ultimately depends on the level of abstraction of the state machine. If, for example, each state transition is triggered by a method call, a static call graph can be used to identify impossible call sequences. If the state machine is at a lower level of abstraction, e.g. transitions correspond to individual blocks of statements, techniques such as symbolic execution can be used to identify impossible sequences of statement blocks [30].
- **Testing** can answer specific questions about software behaviour. Questions may be in terms of the behavioural model (e.g. 'Is it possible to reach state A from state B?'), in which case established model-based testing techniques can be adopted [2]. Alternatively they can be at a lower level, in terms of the source code itself (e.g. 'is it possible to reach statement s at all?'), in which case there are several established structural testing techniques.

- **The developer** will usually be able to provide a limited amount of negative information about program behaviour.

The combination of dynamic analysis with static analysis is not novel in itself, and has been a feature of a number of established analysis techniques. However, conventionally the two analysis types have been seen as complementary in the sense that each covers a subset of program executions [1]. One novel aspect of using static analysis in the manner suggested above is the fact that we do not need it to provide information about feasible program executions (a task for which it is often ill-suited anyway); instead this approach would take advantage of its strengths - namely identifying executions that are *in*feasible.

Active dynamic analysis

Dynamic analysis, particularly in the context of program comprehension, is not necessarily restricted to passive techniques (which expect a single initial set of traces and produce a result in a single step). Active and iterative techniques in grammar inference (see section 3.3.1) have been shown to outperform passive techniques, and are much more usable because they do not require all of the input to be identified and provided at once. One major weakness of such approaches is however the fact that it is difficult to find a reliable oracle to answer the questions that are posed by the technique. If the technique has been provided with only a sparse sample of system behaviour, it may be necessary to ask a large number of questions to obtain all of the missing information. A human can answer a limited number of questions, and is not guaranteed to answer all questions correctly without perfect knowledge of the underlying system behaviour (which is usually unlikely).

The domain of software analysis is ideal for active techniques that require oracles. As discussed above, there are numerous means to *automatically* query the underlying software system about its behaviour, without requiring a substantial amount of manual interference. Questions can be posed as tests, which can be executed with automated testing frameworks, call graphs can be queried, and invariants can be checked automatically.

The key to generating an accurate and efficient active dynamic analysis technique is knowledge of what would constitute a complete set of traces. If the analysis is intended to infer a state machine, it would be reasonable to adopt the notion of a *characteristic* sample from the RPNI work. This states that a model will only be guaranteed to be complete if there is enough positive and negative information to differentiate between states that are not equivalent, and all of the behaviours that are part of the underlying system behaviour are included. In this case, one reasonable questioning strategy (as is the case with QSM for example) would be to gear the question generation process to distinguishing between as many states as possible.

5. Evaluation and Discussion

The previous section presents three principles from the field of grammar inference that can be used to improve dynamic analysis techniques. This section demonstrates the potential contribution of each principle, by measuring the resulting improvement with respect to a large collection of synthesised quasi-random state machines, which are specifically generated to resemble the sorts of state machines that represent software systems. To better evaluate the accuracy of grammar inference / dynamic

analysis techniques, we present and demonstrate an improved evaluation technique that can provide more insights into the type of information returned by a technique. Section 5.1 describes how the synthetic machines and the rest of the experiments are constructed, section 5.2 describes our evaluation approach, and the following subsections describe each experiment in turn.

5.1. Experimental setup

The synthesised state machines are quasi-random. There are particular constraints that must be satisfied for the sake of mimicking real state machines, which are:

- They must be **deterministic** - for every state, there can be no two outgoing transitions with the same labels.
- They must be **minimal** - they must not contain more states than are necessary to accept / reject sequences (as abstracted from program traces). Given a non-minimal machine, there are established techniques that can be used to derive a minimal equivalent [31].
- All of the states must be **reachable** from the initial state - this is implied when a machine is minimal.
- The machine must support the the ability for multiple (differently labelled) transitions to lead from one state to another.

Machines are generated by specifying the number of states, the number of labels (size of the machine alphabet), and a desired number of state transitions. The program executions that form the basis for inference are simulated by taking random paths across the machine. Their length (number of transitions) is chosen to fit a uniform distribution $[0, d + 5]$, where d represents the maximum depth of the machine (this is inspired by Dupont *et al.* [15]). The selection process ensures that no path is entirely subsumed by another path. In the case of negative paths, a length l is picked from the uniform distribution, a positive random path is traced to length $l - 1$ and is then appended by a single input that would cause the string to be rejected. In Dupont's work, as mentioned in the previous experiment, $\frac{|Q|^2}{2}$ is considered to be the upper limit of the number of (initial) traces required to accurately infer some target machine. We instead established that the limit of $4 * |Q|$ produced a sufficiently accurate result, despite the fact that the total number of traces is much smaller as machines increase in size (the total set of traces is reduced to 100 for 25-state machines, whereas it would have been 312 using Dupont's measure).

The experiments use sets of machines where $|Q| = 5$ and $|Q| = 25$. The machines were constructed to represent both complex and simple models. *Crowded* machines represent complex systems, and have $3 * |Q|$ transitions, where $|\Sigma| = |Q|$. *Sparse* machines represent simpler systems, and have $2 * |Q|$ transitions, where $|\Sigma| = 3$.

5.2. Measuring Accuracy with Precision and Recall

Precision and recall [32] is a measure that is conventionally used to measure the accuracy of information retrieval systems. In the context of information retrieval, precision measures the proportion of retrieved information that is relevant (exactness) and recall measures proportion of relevant information that is retrieved (completeness). Precision and recall are computed with respect to two

sets - what is relevant (REL) and what has been retrieved (RET). Precision (proportion of captured information that is required) is computed by $\frac{|REL \cap RET|}{|RET|}$ and recall (proportion of required information that is captured) is computed by $\frac{|REL \cap RET|}{|REL|}$.

When used to evaluate the accuracy of reverse-engineered state machines (as typified by the work of Lo *et al.* [24]) RET and REL are allocated test paths in the machines. REL denotes the set of test paths that are accepted by the specification machine S and RET denotes the set of test paths that are accepted by the hypothesis machine H. The two machines S and H are deemed to be identical if they accept the same strings. This metric only takes the accepting behaviour of two machines into account, but ignores their rejecting behaviour and cannot be used to conclusively establish the equivalence of two machines without potentially requiring an infinite test set. Conventional grammar inference test sets contain an even balance between valid and invalid strings to ensure that the machines both reject and accept the same sets of sequences.

To account for both accepting and rejecting state machine behaviour, the conventional precision and recall measure has to be refined. Instead of computing a single precision and recall tuple, we compute one that describes the accuracy of H in terms of the set of traces it should accept, and the other in terms of how the set of traces it should reject. For this reason, we divide RET and REL into RET^+, RET^-, REL^+ and REL^-.. Thus, $precision^+ = \frac{|REL^+ \cap RET^+|}{|RET^+|}$ and $recall^+ = \frac{|REL^+ \cap RET^+|}{|REL^+|}$, and the same approach is used to compute the negative precision and recall from RET^- and REL^-. The final overall precision and recall values can be computed as the weighted harmonic mean of the positive and negative parts, so $precision = \frac{2 * precision^+ * precision^-}{precision^+ + precision^-}$ and $recall = \frac{2 * recall^+ * recall^-}{recall^+ + recall^-}$. The final precision and recall scores describe in absolute terms the completeness and exactness of a reverse engineered machine in terms of the sets of traces it accepts *and* rejects.

5.3. Experiment 1 - Compressing multiple traces as a PTA

This experiment shows the extent to which a set of multiple traces can be compacted when stored as a PTA. The boxplots in figure 3 show the extent to which random paths through 5- and 25-state random machines are compressed, given different numbers of paths (in terms of percentages of $4 * |Q|$ paths). Each box-plot represents the distribution of percentages (for a given percentage of the total learning sample) to which the set of traces was compressed by storing it as a PTA.

One apparent observation is that the compression rate is higher for sparsely populated machines than it is for crowded ones. This suggests that (abstracted) execution traces become harder to compress with the PTA as software systems under analysis become increasingly complex. This seems intuitive; additional transitions in a machine increase the number of possible paths through it. Accordingly paths are less likely to traverse the same transitions, are more likely to vary and are thus more likely to have distinct prefixes, which implies that the prefix tree containing them would be larger.

The relatively low compression values for the 5 state, 10% sample point can be explained by the fact that only two strings are provided (10% of $4 * |Q|$). If these two paths are completely different (which is quite likely), there will be no branching points in the PTA, and correspondingly no compression. However, as soon as more strings are provided, the likelihood of shared prefixes (and thus branching points in the PTA) is increased, which accordingly results in higher compression rates.

Suprisingly, apart from the 10% and 20% points in the 5 state examples, the rate of compression remains almost constant with a very low deviation as more strings are added to the PTA. As an

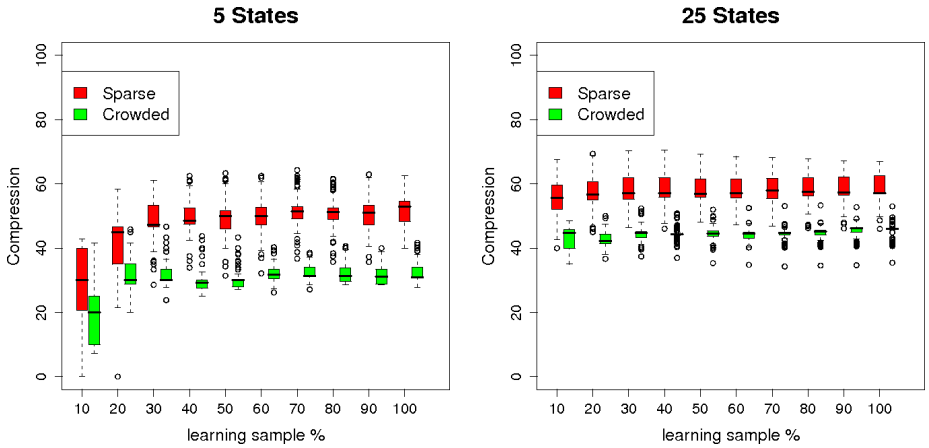
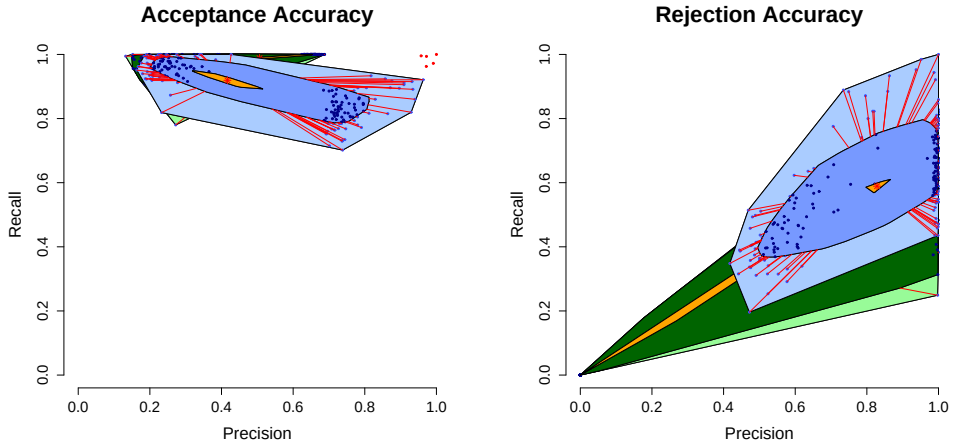


Figure 3. Compression rates for sparse and crowded random state machines with 5 and 25 states

example, the compression rate for sparse 25-state graphs remains virtually constant at around 58% compression, and at around 43% for crowded graphs. The compression rate for 10% of the learning sample is virtually the same as for 100%.

5.4. Experiment 2 - Incorporating negative information

The provision of negative traces is necessary to reverse engineer a precise machine. As with grammar inference, dynamic analyses often have to decide whether two points in a set of traces correspond to the same state, and will only reliably decide that this is not the case if there is enough negative information to differentiate between the two. For this experiment, 100 sparse 25-state machines were used. For each machine the (passive) EDSM state merging algorithm (employing the blue-fringe search technique) was executed on two sets of paths (pos and neg). The pos set was populated with 100 random paths, and the neg set consisted of all of the traces in pos, but specified one impossible input to follow each string in pos. In other words, two negative transitions were added to the leaf nodes of the APTA constructed by pos. Because the strings in pos alone contained no negative information at all, the k limit was set to 3 to prevent it from over generalising (this choice is based largely on intuition - any lower value tended to result in gross overgeneralisations). To make sure that any differences in performance between the negative and the positive samples were not influenced by the k limit, it was kept at 3 for the inference of machines using the negative traces as well. Figure 4 contains two bagplots that compare the accuracy of the two sets of state machines produced with and without a portion of negative (invalid) sequences. The dark bags show the precision-recall spread for machines that are produced without negative sequences, and the blue bags represent the spread for machines that are produced with the help of negative information. Figure (a) shows how accurate the machines are at accepting sequences, and figure (b) shows how accurate they are at rejecting invalid sequences.



(a) $precision^{+}$ and $recall^{+}$ (b) $precision^{-}$ and $recall^{-}$

Figure 4. Plots showing distributions of precision and recall of reverse-engineered state machines

Without the use of negative information, there is a danger that the state-merging algorithm can end up over-generalising. This results in a machine that accepts too many sequences that should in fact be rejected. This is illustrated in figure (a), where the positive-only machines have a high recall (they accept all of the strings that they need to), but a low precision (generally below 50% - they also accept too many sequences that should be rejected). This is confirmed by figure (b), where machines that are constructed with no negative information can have a low precision and recall (they accept too many sequences that they should reject). In some cases, the bag extends down to zero precision and zero recall, which means that the machine does not accurately reject any sequence at all.

The machines that are produced with negative sequences are usually more successful in both respects. In both figures (a) and (b) the results for machines constructed with negative strings tend to be in two clusters. Machines with a relatively low precision and recall are the result of training sets where there was not enough negative information to prevent a false merge from occurring at some point during the merging process. The machines that produce a relatively high precision and recall value are the result of training sets where there was sufficient negative information to prevent such merges. If a false merge occurs early on in the sequence of state merges, subsequent merges will merely compound the error, resulting in a state machine that is highly inaccurate.

The experiment illustrates that positive sets of traces showing valid system behaviour can only be used to accurately infer system behaviour if they are complemented by a sufficient amount of information about invalid behaviour as well. This information need not necessarily be supplied in the form of program traces, but could also be synthesised by using alternative sources of information such as static analysis. Nonetheless, there remains the problem of identifying *what* negative information is

required. Exhaustively supplying every infeasible execution trace would be impractical; the developer needs to be guided to only supply information that is relevant to the inference of the final state machine.

5.5. Experiment 3 - Active dynamic analysis

The notion of active dynamic analysis, where the analysis elicits only the information it requires from the software system (and / or its developer), addresses the aforementioned problem. The use of active techniques in grammar inference is becoming increasingly popular, because they produce results that are more accurate, are more resilient to sparse sets of input samples [17, 15] and better at identifying states that are particularly hard to reach [33]. This experiment illustrates the performance of such an active technique (Dupont's QSM technique [17, 15]) versus the passive state merging technique used for the previous experiment. The idea of using QSM to reverse engineer state machines by dynamic analysis has been previously presented by the authors [34].

Figure 5 shows the precision and recall distributions of the active and passive algorithms for (crowded) 5-state and 25-state machines. The passive algorithm was used in the same vein as the previous experiment (provided with negative traces, and the k limit was set to 1). For the active algorithm the k limit was set to zero.

In all experiments, the active algorithm outperformed the passive version. Passively learned machines have a very low recall (fail to capture most of the required machine behaviour) and mediocre precision (much of the behaviour that *is* captured is incorrect). Nonetheless, the active learner is relatively successful at using this information to garner further information, which is successful at increasing both precision and recall. Given 100% of the input traces (i.e. all $4 * states$ traces) the actively learned machines could generally be considered as accurate (the median was above 95% precision and recall).

The improved performance of the active algorithm is not surprising, as it benefits from the ability to collect the information it needs, whereas the passive algorithm has to work with what it's given. However, an obvious factor in the use of the active algorithm is its cost - how many questions does it have to ask of the system to produce an accurate result? The boxplots in figure 5 clearly show that the number of questions rises as the number of traces increases. Given the full set of 100 initial traces, the median for the active algorithm is at 2313 questions, but can range as high as 8000.

The high number of questions is primarily due to the complexity of the target machine (large number of transitions and possible labels). In a conventional grammar inference context (where a human is providing the input), this would clearly render the technique impractical. However, as was mentioned in section 4.2, answering these questions can be mostly straightforward in the context of reverse engineering, due to the availability of automated static analysis techniques. Any questions that cannot be answered by them can be rephrased as tests. Investigating to what extent the question answering can be automated in this vein forms a major part of our future work.

Although active dynamic analysis can help to identify the set of executions that are necessary for a more complete analysis, there is still the inevitable problem that the process can in practice still be prohibitively expensive for large numbers of questions (even with the use of static analysis and automation of the question answering). So far emphasis has been placed on simply establishing the improvement in accuracy and reliability that is possible with active dynamic analysis - without taking the practicality of the technique into account. We conclude this experiment by demonstrating that, by

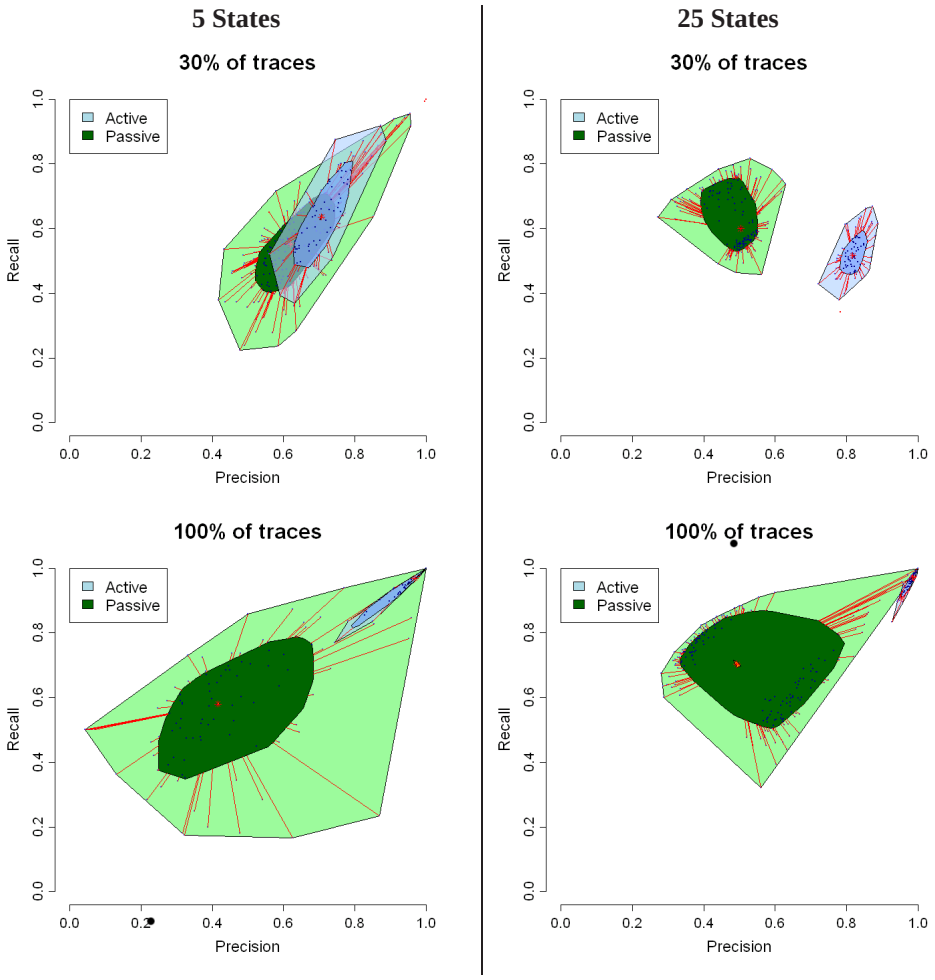


Figure 5. Precision and recall for passive and active algorithms for machines with 5 and 25 states

adding constraints to the QSM algorithm, it is however possible substantially reduce the number of questions and in doing so only slightly reduce the accuracy of the final result.

With Dupont's unconstrained algorithm, a large number of questions ask questions about merges that are almost certain to be correct anyway (the merges have a high score). In practice, questions are only necessary if there is not enough information there to support the correctness of the merge. One possibility therefore is to only ask questions if the score for a particular merge is below a given threshold, and to accept any proposed merges above that threshold as correct by default. The number

of questions is reduced dramatically - the number of questions for 100% of the traces reduced from a mean of 3824 questions to 220. However, despite this substantial reduction in the number of questions, the mean reduction in precision and recall for 100% of the traces is only by 13% and 6% respectively. Whether this reduction is acceptable ultimately depends on the application of the dynamic analysis (for many areas it would certainly be acceptable). Even with such a reduction in precision and recall, it can be argued that these results are still a much more accurate, and therefore appealing alternative to the conventional passive dynamic analysis techniques discussed in section 2.

6. Conclusions and Future Work

This paper has presented the parallels between the two fields of grammar inference and dynamic analysis. Although the problems of dynamic analysis and grammar inference have been largely addressed separately, there is enough of an overlap to adopt solutions from one field to the other. This paper shows, from the perspective of dynamic analysis, at the field of grammar inference, how some of the techniques that have proved successful in the field of grammar inference can just as well be applied to dynamic analysis techniques.

To demonstrate the value of adopting these grammar inference techniques in the context of dynamic analysis, a number of small experiments have been carried out. We have developed an algorithm that synthesised random state machines with characteristics of software state machines. We have generated these in their hundreds, varying the number of states transitions, to show empirically how analysis algorithms employing particular techniques that are used in grammar inference outperform the standard approaches that are used for dynamic analysis.

Although the benefits are apparent, these can only be achieved by changing the process of dynamic analysis itself. This paper suggests greater use of negative information, as well as iterative (active) approaches that test and probe the program during the analysis process. It also suggests that complementary techniques such as static analysis can play a greater role in the identification of negative information about program behaviour.

Our future work will focus on minimising any required manual input (such as tests and traces) whilst retaining the accuracy of the final state machine. Ultimately, we aim to implement a fully automated technique, that automatically tests and analyses the system, and can produce a correct and accurate state machine. This will only be possible by adopting and extending the techniques from grammar inference that we have demonstrated to be so effective in this paper.

ACKNOWLEDGEMENTS

The authors thank the anonymous referees for their valuable comments.

REFERENCES

1. Ernst M. Static and Dynamic Analysis: Synergy and Duality. *Proceedings of the International Workshop on Dynamic Analysis (WODA'03)*, 2003.
2. Lee D, Yannakakis M. Principles and Methods of Testing Finite State Machines - A Survey. *Proceedings of the IEEE*, vol. 84, 1996; 1090–1126.

3. Angluin D, Smith CH. Inductive inference: Theory and methods. *Computing Surveys* 1983; **15**(3):237–269.
4. Parekh R, Honavar V. *The Handbook of Natural Language Processing*, chap. Grammar Inference, Automata Induction and Language Acquisition. 2000; 727–764.
5. Gold E. Language identification in the limit. *Information and Control* 1967; **10**:447–474.
6. Chomsky N. Three models for the description of language. *IRE Transactions on Information Theory* 1956; **2**(3):113–124.
7. Nerode A. Linear automata transformations. *Proceedings of the American Mathematical Society* 1958; **9**:541–544.
8. Gold E. Complexity of automaton identification from given data. *Information and Control* 1978; **37**:302–320.
9. Kearns M, Valiant L. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM* 1994; **41**.
10. Trakhtenbrot B, Barzdin Y. *Finite Automata, Behavior and Synthesis*. North Holland: Amsterdam, 1973.
11. Lang K. Random DFA's can be approximately learned from sparse uniform examples. *COLT*, 1992; 45–52.
12. Oncina J, Garcia P. Inferring regular languages in polynomial update time. *Pattern Recognition and Image Analysis*, vol. 1. 1992; 49–61.
13. Garcia P, Cano A, Ruiz J. A comparative study of two algorithms for automata identification. *Proceedings of the International Colloquium on Grammar Inference (ICGI'00)*, Lecture Notes in Computer Science, 2000.
14. Oncina J, Garcia P, Vidal E. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Trans. Pattern Anal. Mach. Intell* 1993; **15**(5):448–458.
15. Dupont P, Lambeau B, Damas C, van Lamsweerde A. The QSM algorithm and its application to software behavior model induction. *Applied Artificial Intelligence* 2008; **22**:77–115.
16. Dupont P. Incremental regular inference. *Proceedings of the International Colloquium on Grammar Inference (ICGI'96)*, 1996.
17. Damas C, Lambeau B, Dupont P, van Lamsweerde A. Generating annotated behavior models from end-user scenarios. *IEEE TSE* 2005; **31**(12):1056–1073.
18. Lang K, Pearlmutter B, Price R. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. *Proceedings of the International Colloquium on Grammar Inference (ICGI'98)*, vol. 1433, 1998; 1–12.
19. la Higuera CD, Oncina, Vidal. Identification of DFA: Data-dependent vs data-independent algorithms. *ICGI: International Colloquium on Grammatical Inference and Applications*, 1996.
20. Angluin D. Learning regular sets from queries and counterexamples. *Information and Computation* 1987; **75**:87–106.
21. Biermann A, Feldman J. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers* 1972; **21**:592–597.
22. Ammons G, Bodík R, Larus J. Mining specifications. *POPL'02*, Portland, Oregon, 2002; 4–16.
23. Cook JE, Wolf AL. Discovering models of software processes from event-based data. *ACM TOSEM* 1998; **7**(3):215–249.
24. Lo D, Khoo S. QUARK: Empirical assessment of automaton-based specification miners. *WCRE*, IEEE Computer Society, 2006; 51–60.
25. Lo D, Khoo S. SMArTIC: towards building an accurate, robust and scalable specification miner. *SIGSOFT FSE*, 2006; 265–275.
26. Lorenzoli D, Mariani L, Pezze M. Inferring state-based behavior models. *Proceedings of the International Workshop on Dynamic Analysis (WODA'06)*, 2006.
27. Reiss S, Renieris M. Encoding program executions. *ICSE*, IEEE Computer Society, 2001; 221–230.
28. Li K, Groz R, Shahbaz M. Integration testing of distributed components based on learning parameterized I/O models. *Proceedings of FORTE'06, Lecture Notes in Computer Science*, vol. 4229, Springer, 2006; 436–450.
29. Hungar H, Niese O, Steffen B. Domain-specific optimization in automata learning. *International Conference on Computer Aided Verification (CAV'03)*, 2003.
30. Walkinshaw N, Bogdanov K, Ali S, Holcombe M. Automated discovery of state transitions and their functions in source code. *Software Testing, Verification and Reliability* 2008; **18**(2).
31. Hopcroft J, Ullman J. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
32. Rijsbergen CJV. *Information Retrieval*. Butterworth-Heinemann: Newton, MA, USA, 1979.
33. Bongard J, Lipson H. Active coevolutionary learning of deterministic finite automata. *Journal of Machine Learning Research* 2005; **6**:1651–1678.
34. Walkinshaw N, Bogdanov K, Holcombe M, Salahuddin S. Reverse engineering state machines by interactive grammar inference. *WCRE'07*, 2007.