Reverse Engineering State Machines by Interactive Grammar Inference

Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, Sarah Salahuddin
Regent Court, 211 Portobello Street, Sheffield S1 4DP
E-mail: {n.walkinshaw,k.bogdanov,m.holcombe,s.salahuddin}@dcs.shef.ac.uk

Abstract

Finite state machine-derived specifications such as X-Machines, Extended Finite State Machines and Abstract State Machines, are an established means to model software behaviour. They allow for comprehensive testing of an implementation in terms of its intended behaviour. In practice however they are rarely generated and maintained during software development, hence their benefits can rarely be exploited. We address this problem by using an interactive grammar inference technique to infer the underlying state machine representation of an existing software system. The approach is interactive because it generates queries to the user as it constructs a hypothesis machine, which can be interpreted as system tests. This paper describes (1) how an existing grammar inference technique (QSM) can be used to reverse-engineer state-based models of software from execution traces at a developer-defined level of abstraction and (2) how the QSM technique can be improved for a better balance between the number of tests it proposes and the accuracy of the machine it derives. The technique has been implemented, which has enabled us to present a small case study of its use with respect to a real software system, along with some preliminary performance results.

1. Introduction

Perceiving software as a state machine enables a developer to design, document and rigorously test a program in terms of its behaviour. States characterise the behaviour of the system at a particular point in its execution and transitions, which are triggered by system functions, lead from one state to another. Despite their apparent benefits, the wide-spread adoption of state machines in software engineering has been hampered by the fact that they are rarely produced and maintained in the first place. This is usually due to the facts that (1) software development is usually conducted under restrictive time constraints and (2) the dynamic nature of software evolution makes it difficult to keep designs and documentation up to date. As a result, the rigorous (and potentially automated) state machine-based testing techniques can rarely be exploited and are instead replaced by inferior ad-hoc techniques.

Grammar inference aims to identify the grammar for a language, given a sample of word sequences that belong to and (optionally) do not belong to the language. In practice grammar inference has a broad range of applications beyond its original field of natural language acquisition. A regular language can be represented by a finite state machine, and the provided sample of sequences can be interpreted as sequences that are either accepted or rejected by that machine. Because state machines can model the behaviour of large variety of systems besides regular language grammars, grammar inference techniques can be used to identify the underlying state transition structure of *any* system that can be represented as a finite state machine.

Previous software engineering research recognises the apparent link between the aforementioned absence of state machine models of software systems and the possibility of generating them automatically with grammar inference techniques. Dupont *et al.*'s QSM state-merging approach [6, 8] takes as input an initial sample of manually generated scenarios from the user, and uses these as a basis to interactively generate a state machine of the system. The QSM approach is primarily intended to help automating the generation of (web-) applications, where it is reasonable to expect that the developer can be coerced into manually generating a set of scenarios. In our case however, where we presume that the system has already been developed and the specification serves the purpose of testing and inspection, it is unrealistic to expect the user to (a) have developed any specifications in the first place, and (b) have kept them up to date. Hungar *et al.*'s approach [14, 13] is more suitable in that respect; it takes a set of program traces as input and uses an optimised version of Angluin's L* algorithm [1] to infer the machine. The use of program traces eliminates the need for an initial set of manual scenarios because these are generated automatically from a set of program executions. However, the L* technique by itself requires a substantial amount of guidance from the developer and the optimisations they employ to reduce the guidance are tied to a rel-

atively specific application domain (they take advantage of symmetry and partial-order properties that arise from the distributed and concurrent nature of their phone system).

This paper builds on Hagerer *et al.*'s idea of using execution traces as an input for grammar inference techniques. However instead of using Angluin's L* technique, it uses Dupont *et al.*'s QSM technique which, by placing stronger requirements on the set of initial samples than Angluin's technique, requires less guidance and has thus been shown to scale relatively well to systems with large numbers of states. The paper presents a number of improvements to Dupont *et al.*'s technique, which further reduce the amount of guidance required. The main contributions of this paper are as follows:

- A tractable approach to reverse engineering state machines from execution traces.
- The integration of the technique into a testing framework; whereas the QSM technique conventionally relies upon a human oracle to answer questions about the system under analysis, in our context these questions can be posed as system tests.
- A number of improvements of the QSM technique, which improve the ability to generate more accurate machines with less input.
- An implementation of the technique, along with a case study that shows how to reverse engineer a state machine from the open-source JHotDraw drawing framework.

Besides the case study, we have obtained some preliminary results that investigate the scalability of our technique to inferring large state machines. These show that, at least for the class of machines considered herein, the technique scales to relatively large systems without requiring a substantial increase in the amount of input from the developer or system tests.

2. Regular Grammar Inference and the QSM Technique

This section introduces the problem of grammar inference, along with some of the approaches that have been used to address it in the past. A more detailed introduction is provided of the state merging techniques, specifically the QSM technique which we have used in our implementation.

2.1. The Grammar Inference Problem and its Solution

The problem of grammar inference, as first identified by Gold in his fundamental paper [10], is to identify a language L given a set of sample sequences 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 $S^+ \subseteq L$ and $S^- \subseteq \Sigma^* \setminus L$. When L is a regular language, its grammar can be represented by a deterministic finite automaton (DFA) and the problem can be re-interpreted as one that aims to identify the DFA that produces L. This is a particularly suitable representation for inference algorithms, because of the fact that (a) there exists a single minimal DFA for any regular language, and (b) there exist a host of efficient DFA algorithms for tasks such as minimisation and removing non-determinism [12].

A number of techniques have been developed that attempt to solve the above problem. Angluin's L* technique [1] for example uses S^+, S^- and Σ to construct queries of various types that systematically explore the target system's behaviour, and in the process constructs a complete DFA for the system. This has successfully been applied to a variety of software-engineering problems (c.f. [14, 13, 18]), but is often limited by the demands it places on the user. Because of the fact that it will always systematically infer the complete automaton, there is no upper bound on the potential number of membership queries, which can in practice render the technique impractical. As an example, Hungar *et al.* [13] observed that without their domain-specific optimisations, the number of queries for a four state system using L* was 108, which rose to 15425 for 28 states.

2.2. State Merging

An alternative approach to Angluin's $L*$ technique is to shift the emphasis from the oracle that provides input during the learning process to the quality of the initial set of sample sequences S. State merging techniques require less guidance from an oracle, but can still produce an accurate state machine on the condition that the initial set of samples S covers enough of the target machine. If this is the case, the process of generating the hypothesis machine simply consists of generalising from set of samples that has been provided, without working out which elements of machine behaviour might have been missed out. Because the process is purely one of generalisation, the number of membership queries is bounded by the size of the initial sample S.

State merging techniques usually take as input an 'augmented prefix tree automaton' (APTA) constructed from S, where the common prefixes for each sequence lead to the same branch node, and every unique suffix leads to a leaf node. Nodes are labelled as either positive or negative, depending on whether the corresponding sequence belongs to S^+ or S^-. Figure 1 illustrates the APTA for a set of sequences that correspond to executions of a simple text ed-

$$
\begin{aligned}
S^+ = \quad & \{< load, edit, edit, save, close >, \\
& < load, edit, save, close >, \\
& < load, close, load >\} \\
S^- = \quad & \{< close >\}
\end{aligned}
$$

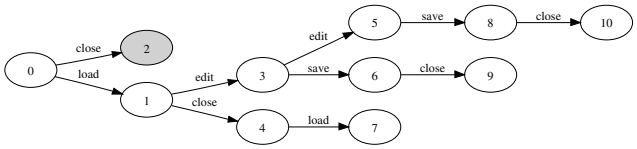


Figure 1. Augmented prefix tree automaton with associated sequences

itor. The APTA is not minimal and presumably, since S is only a sample of the target language, does not correctly accept or reject every sequence in the L. Consequently it is the algorithm's task to generalise and minimise the APTA by strategically selecting nodes to merge. One inherent danger is that the merging process might be over-zealous, resulting in a machine that is too general, accepting too many sequences that would be rejected by the actual system.

State merging algorithms are guaranteed to produce the target automaton if the training data is exhaustive (i.e. contains all input sequences up to a sufficient size). Their popularity is however due to the fact that they also produce reasonably accurate results for incomplete or sparse sample sets [15]. In fact, if the training set is not exhaustive, the target machine will still be produced if the training data is *characteristic* of the machine [20], i.e. there is enough training data to reach every state, as well as differentiate between any pair of non-equivalent states.

2.3. The QSM Algorithm

Dupont *et al.*'s QSM algorithm [8] is a state-merging algorithm with two features that make it particularly appealing. It employs a sophisticated search mechanism to select candidate nodes to merge, and prevents over-generalisation by posing membership queries to the end-user whenever the resulting machine may accept or reject sequences that have not been ratified by the user. The pairs of merge candidate states are selected by using the Price's 'Blue Fringe' state merging algorithm [16]. Every time a new hypothesis machine is generated, any strings that are accepted or rejected by the new machine and are not handled by the previous machine are posed as questions for the user to ratify.

Figure 2 provides both the QSM algorithm and and Price's state selection algorithm. The algorithm is based on Dupont *et al.*'s description [8], which can be referred to for further details. Given two sets S^+ and S^- it starts by

Algorithm QSM

Input Sets of accepted and rejected sequences (S^+, S^-)

Output A minimal DFA A consistent with an extended collection (S^+, S^-)

Uses
$initialState(A)$ Returns the initial state of A
$generateAPTA(S, S')$ Generates an augmented prefix tree acceptor from S and S'
$selectStatePairs(A)$Uses the Blue-Fringe EDSM approach to select merge candidates from A (see below)
$merge(A, q, q')$Merges nodes q and q' in A, and ensures that the resulting machine is deterministic
$compatible(A, S^+, S^-)$Checks that$A$ is consistent with $S^+$and S^-
$generateQuery(A, A')$ Returns a set of questions over A' that cannot be classified as accepted or rejected by A
$checkWithEndUser(query)$Returns true if the query is accepted, and false otherwise

Declare $Test$: sequence in A which (in this context) corresponds to a test sequence

```
A ← generateAPTA(S+, S−)
init ← initialState(A)
foreach (q, q′) ← selectStatePairs(A) do
  A′ ← merge(A, q, q′)
  if compatible(A′, S+, S−) then
    foreach query ← generateQuery(A, A′) do
      if checkWithEndUser(query) then
        S+ ← S+ ∪ query
      else
        S− ← S− ∪ query
        return QSM(S+, S−)
  A ← A′
return A
```

Algorithm $selectStatePairs$

Input Automaton A, initial state $init$

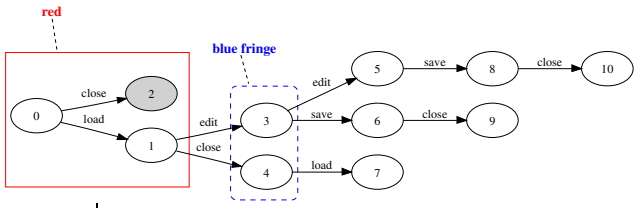
Output Ordered set $Pairs$ of state pairs ordered by their score

Uses
$computeBlue(A, R)$returns the set of states that are neighbours of states in R in A but do not themselves belong to R
$computeScore(A, q, q')$Computes the suffixes for q and q' and increments the score for every overlapping transition label in the two suffixes. Returns $-\infty$ if the target states for any overlapping suffix transitions have different labels ($+$ or $-$) because this implies an incompatible merge.

Declare $Blue$: set of states
$score$: integer
$mergable$:boolean

```
Pairs ← ∅
R ← init
foreach blue ← computeBlue(A, R) do
  mergable ← false
  foreach r ∈ R do
    score ← computeScore(A, blue, r)
    if score ≥ 0 do
      mergable ← true
      Pairs ← Pairs ∪ (score, (r, blue))
    if ¬mergable
      R ← R ∪ blue
return Pairs
```

Figure 2. QSM and merge candidate selection algorithms



Pair	(1,3)	(0,4)	(0,3)	(1,4)	(2,3)	(2,4)
Score	3	1	0	0	-∞	-∞

Figure 3. Illustration of the blue fringe with respect to the APTA in figure 1

constructing the augmented PTA[1] (see figure 1). It invokes Price's Blue-Fringe $selectStatePairs$ function, which returns a set of state pairs that are ordered according to their suitability (this is elaborated below). The QSM algorithm iterates through the candidate pairs in the order of their score (starting with the highest scoring pair), and calls the $merge$ function. Merging a pair of states may result in a non-deterministic machine, so the $merge$ function is also responsible for ensuring that a non-deterministic machine is transformed into a deterministic one. If the resulting machine remains compatible with $S^+$and S^-(checked by the $compatible$ function), the $generateQuery$ function produces a set of sequences that do not belong to S^+ or S^-, but are part of the language of the new machine. In the context of Dupont *et al.*'s application, these are posed to the end-user (as will be described in section 3, these queries can be formulated as tests for the implementation itself). If the query is accepted, the test sequence is added to S^+, the input machine is updated to the merged version, and the algorithm loops onto the next pair of candidate states. If it is unsuccessful, the merge is invalidated by adding the query sequence to $S^-$and QSM is called recursively with the extended S^+ and S^-sets.

As mentioned previously, the $selectStatePairs$ is responsible for generating an ordered set of candidate nodes to be merged. An effective selection strategy is key to the efficiency of the algorithm. Conventional state merging algorithms such RPNI (Regular Positive Negative Inference) [20] would order candidate node pairs in their breadth-first order, so for figure 1 the order of possible merges would be $\{(0,1),(0,2),(0,3)...(1,3),(1,4),(1,5)...\}$. A number of much improved pair selection strategies have been developed, one of which is the 'Blue-Fringe' technique [16], which works by restricting the set of candidate pairs and applying a heuristic to order them in terms of their suitability.

[1]It should be noted that Dupont initially present the QSM algorithm as using just the non-augmented PTA, which is based only on positive samples. However, if the Blue-Fringe $selectStatePairs$ is used, it requires a tree that is constructed from both positive and negative samples (see Lang *et al.* [16] and Dupont *et al.* [8] for details).

The blue-fringe technique tags states that cannot be merged as red, and tags all adjoining non-red states as blue. The aim is to identify and merge the most suitable pairs first, to minimise the possibility of carrying out an invalid merge. The order in which pairs of states are merged is decided by first evaluating every possible pair of red-blue merges via the $computeScore$ function. If a blue state cannot be merged with any red state ($computeScore$ returns a negative value), it is upgraded to a red state and the process is repeated. Figure 3 shows the pool of merge candidates along with their scores for the APTA in figure 1. Pair $(1,3)$ has the highest score, because a path $< edit, save, close >$ can be followed from both of them; the merging algorithm would attempt this pair first, because the available evidence suggests that they are most likely to be equivalent states. Pairs $(2,3)$ and $(2,4)$ receive negative scores because the merging process entails an accepting state to be merged with a non-accepting one (leading to a contradiction with S^-). For further details on the Blue-Fringe algorithm and results on its performance in comparison with other state merging strategies, the reader is referred to Lang *et al.*'s original paper [16] and Dupont *et al.*'s QSM paper [8].

3. Using QSM with Dynamic Analysis to Infer State Machines

The QSM approach has so far primarily been applied in a software requirements engineering context, which does not presume the availability of an existing implementation. This work applies the QSM technique to the problem of reverse-engineering a specification from its implementation. The intrinsic benefit of applying the QSM technique in a reverse engineering environment is the fact that it does not rely as much on human input. Whereas Dupont *et al.*'s work [6, 8] relied upon the end-user to manually generate an initial set of scenarios and to correctly answer the ensuing membership queries, the technique presented herein allows the user to initially provide a set of traces. Answering membership queries is also less error-prone, because the query can be answered by directly executing the system. This section describes the technique in detail, and provides an overview of our implementation.

The state machines considered in this paper form the basis of a number of established state-based modelling techniques, such as Extended-Finite State Machines (EFSMs), X-Machines [11], and Abstract State Machines [3]. Although this technique does not reverse engineer the data constraints that are required to generate a complete specification, it does generate the underlying state transition structure, which suffices for some of their most useful applications, such as test set generation. The underlying state machine can be represented as a labelled transition system (LTS), which can be represented as a 4-tuple (Q, Σ, δ, q_0),

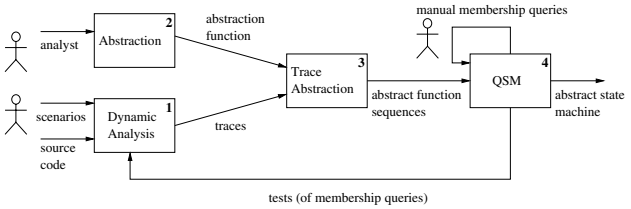


Figure 4. Combining dynamic analysis with QSM

where Q is the set of states, Σ is the set of inputs, δ is the transition function $Q \times \Sigma \rightarrow 2^Q$ and q_0 is the initial state. These modelling techniques can scale to complex systems by defining a set of system functions, which can encapsulate potentially complex system behaviour. Instead of labelling transitions with simple symbols, these techniques label the transitions with the abstract processing functions instead. Consequently, a series of transitions across the graph (as might be specified by a test sequence for example) corresponds to the sequential execution of a series of abstract system functions.

3.1. Technique

The process of using QSM to reverse engineer a state machine from a software system is shown in figure 4. The four main activities along with their respective inputs are described in detail below:

1. **Dynamic analysis:** This activity is responsible for generating a collection of system traces. The system is exercised according to a selection of end-user scenarios that are supplied by the analyst. Depending on the desired level of abstraction, the resulting traces can either be sequences of method invocations or statements. For this paper and our implementation, by execution trace we mean a sequence of method invocations.

2. **Abstraction:** Ultimately, a reverse engineered state machine will only be of practical use if it is constructed at a level of abstraction that can be readily interpreted by the analyst. The abstraction process serves to generate a function that takes as input a low-level program trace (produced by activity 1) and produces as output an equivalent sequence of abstract functions.

3. **Trace abstraction:** This activity applies the abstraction function identified in step 2 to the set of traces generated by step 1. This results in a set of abstract function sequences that serve as input for the following step.

4. **QSM:** In this activity the QSM algorithm presented in section 2 is applied to the sequences of abstract functions. One of the benefits of the availability of an existing implementation is the fact that it is no longer the sole responsibility of the end-user to act as an oracle. The version presented herein alters the behaviour of $checkWithEndUser$. Now, $checkWithEndUser$ interprets membership queries as either tests for the implementation or queries that can be answered manually. If interpreted as a test a trace can be generated from a membership query. If its abstracted sequence of functions matches the membership query, it can be added to S^+, otherwise it is added to S^-.

3.2. Implementation

Our implementation consists of three components: dynamic analysis (activity 1) is carried out with the Eclipse Test and Performance Tools Platform (TPTP)[2], abstraction and trace analysis (activities 2 and 3) are carried out by a single abstraction component and the QSM algorithm (activity 4) is implemented by dedicated QSM component. While the TPTP tool could be used as-is, the other two components were implemented by the authors.

Dynamic analysis with TPTP The Eclipse TPTP framework provides a range of tools for the dynamic analysis of Java programs. Program executions can be monitored at various degrees of granularity. We use TPTP to record program traces as sequences of method invocations, which are recorded to an XML file.

Abstracting from traces In this implementation, an abstract function is identified as a pattern of method calls that occur in the trace. As an example, in the JHotDraw drawing framework from our evaluation, if the `PaletteButton.mouseReleased` method is followed at some point by the `TextTool.activate` (in the same call-stack), we can map that to the `activate_text_tool function`. Depending on the desired level of abstraction a method-level trace will not be sufficiently granular (i.e. if the execution of two abstract functions is distinguished by two branches through the same method). If this is the case, the trace and abstraction functions have to be generated at a statement-level [22]. Each trace is abstracted in turn. Every time a sequence of method calls is identified that belong to an abstract function (they needn't be consecutive), the name of the function is inserted into the abstracted trace.

The QSM component The QSM component takes as input a set of positive function sequences (S^+), and a set of

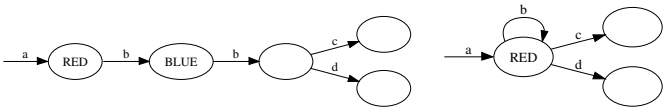
[2]http://www.eclipse.org/tptp/

rejected sequences (S^-). As illustrated in figure 4, there are two ways to answer membership queries (i.e. of identifying whether a sequence belongs to the final state machine or not). Dupont *et al.*'s approach relies on the user to manually state whether or not the sequence is accepted. Because we can also treat the software implementation as an oracle, our implementation provides two means to answer membership queries. For a given query, the user can either state explicitly whether or not the sequence is accepted , or there is the option to obtain an answer by means of testing the implementation. If the user opts to answer the membership query by testing the system, they are asked to provide an XML file (from TPTP) that corresponds to an execution of the sequence of abstract functions represented by the membership query. The XML file is abstracted (step 3 in figure 4) and the resulting abstract sequence of functions is checked to identify whether it contains the membership query as a subsequence. If the membership query is not a subsequence of the trace, the query is answered negatively. Ultimately, we look to fully automate this testing step for simple function sequences.

During our implementation of the QSM algorithm, we have added a number of improvements over the original algorithm, which improve its accuracy in a number of ways. These are summarised below:

- Our implementation uses an improved question generation technique. The aim of the questions is to, for a pair of candidate nodes that are to be merged, identify possible paths in the resulting machine that are not explicitly accepted or rejected by the current machine. The conventional QSM technique constructs a list of questions by concatenating a prefix of the red node with all of the suffixes of the blue nodes in the current machine. However, this does not account for edges that may appear in the hypothesis machine once the machine has been minimised (by merging) and non-determinism has been removed, and can result in invalid questions. Our improved approach generates the questions *after* the merge has been computed, by taking the prefix to the red node with any suffixes of the merged state. The technique results in a more comprehensive set of questions and is illustrated in figure 5.

- Once the QSM algorithm is complete the resulting state machine may still contain invalid edges. Our implementation enables the user to add negative sequences that eliminate them. This can be achieved by selecting a faulty edge $a \rightarrow b$. The implementation then constructs a list of sequences by taking the prefix of a and concatenating it with all non-negated suffixes from b. The user can select an invalid sequence, which is added to $S^-$and the algorithm is restarted.



pre-merge

post-merge

$Q = \{abc, abd\}$

Conventional set of questions

$Q = \{abc, abd, ac, ad\}$

Improved set of questions

Figure 5. Illustration of a merge, along with the questions generated for the hypothesis machine

4. Evaluation

We use the JHotDraw framework as a case study because it is openly available and has been used as the basis for a number of other case studies. In this study we show how to reverse-engineer its underlying state transition structure with respect to a specific set of abstract functions, along with their mappings to sequences of method calls. Section 4.2 takes a more systematic approach to evaluating the accuracy of the technique. The purpose of this section is to show that the process of identifying a state machine using our technique is feasible and practical, and also to highlight some of the issues that arise.

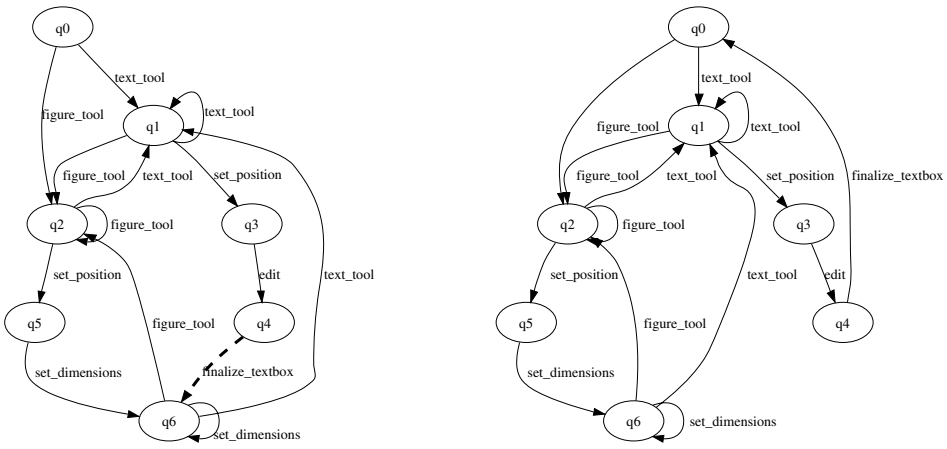
4.1. Case Study

JHotDraw[3] is a well established, open-source Java framework for constructing drawing tools. HotDraw was originally developed in the eighties by Cunningham and Beck as a Smalltalk drawing editor framework. It was then rewritten for Java as JHotDraw, and serves as a showcase to illustrate the use of object-oriented design patterns. JHotDraw is particularly suitable as a case study because it has been extensively explored in other software engineering research projects.

In this case study we concentrate on the specific behaviour of the JavaDraw application, that is included as an example application in the JHotDraw release. A simple state machine of its high-level behaviour, along with a screen shot of JavaDraw is shown in figure 6. It details the processes of either adding a figure or a text box to the figure.

To reverse engineer the machine we start by extracting program traces from JHotDraw and constructing a set of abstraction functions (in any order). The abstractions from sequences of methods to abstract functions are shown in the

[3]http://www.jhotdraw.org



text_tool	<PaletteButton.mouseReleased, TextTool.activate>
figure_tool	<PaletteButton.mouseReleased, CreationTool.activate>
set_position	<CreationTool.mouseDown>
edit	<JTextComponent.caretPosition>
finalize_textbox	<TextTool.deactivate, TextTool.endEdit, TextFigure.setText>
set_dimensions	<CreationTool.mouseDrag>

Figure 6. JavaDraw hypothesis machines and the abstractions

table in figure 6. In theory, in order to guarantee that the resulting machine is an exact match, the user would be expected to provide a set of inputs that is characteristic (i.e. covers every transition and differentiates between every pair of non-equal states). In practice this is not always going to be the case, and the QSM questions are designed to elicit as much of the missing characteristic set as possible. For the case study the system has been executed four times, tracing the method calls with TPTP in the process. The resulting XML files are processed in turn, and result in the following abstract function sequences:

t_1= <figure_tool, figure_tool, set_position, set_dimensions, set_dimensions, set_dimensions, set_dimensions, figure_tool, set_position, set_dimensions>

t_2= <figure_tool, figure_tool, set_position, set_dimensions, set_dimensions, set_dimensions, text_tool, set_position, edit>

t_3= <text_tool, text_tool, set_position, edit, finalize_textbox, text_tool>

t_4= <text_tool, text_tool, set_position, edit, finalize_textbox, figure_tool>

The QSM algorithm, which was modified according to section 3.2, poses 44 queries. An example of such as query might be:

Is the sequence <text_tool, set_position, ***set_dimensions,*** *set_dimensions, set_dimensions, text_tool, set_position, edit> possible?*

This query should be rejected at the first occurrence of set_dimensions, because the process of setting dimensions only applies to figures in JHotDraw, not text boxes. This information can either be supplied directly by the user if they have sufficient knowledge of the underlying system, or it can be posed as a system test. In the latter case we attempt to execute the query and generate another trace with TPTP in the process. The trace is then supplied to our tool, which checks whether the query is a subsequence of the supplied trace or not, and answers the query accordingly.

The resulting machine is shown in figure 6 (a). Closer inspection reveals that, although the machine would correctly accept all of the sequences of abstract functions, it would also accept sequences that should be rejected in practice. This is due to the fact that none of the questions managed to elicit a sequence of functions showing that the edge $q_4 \overset{finalize\,textbox}{\rightarrow} q_6$ is not valid, because it cannot be followed by the function `set_dimensions`. When such an edge is observed once the learning process has finished, the faulty edge can be selected, and the user is presented with a list of all apparently valid edge sequences (by taking the shortest prefix of q_4 and concatenating all suffixes of q_6). Some of these should be invalid, such as: *<text_tool, text_tool, set_position, edit, finalize_textbox,* ***set_position****>*. When a sequence is selected from this list, it is added to S^- and the algorithm is restarted, finally resulting in the correct machine, which is depicted in figure 6 (b).

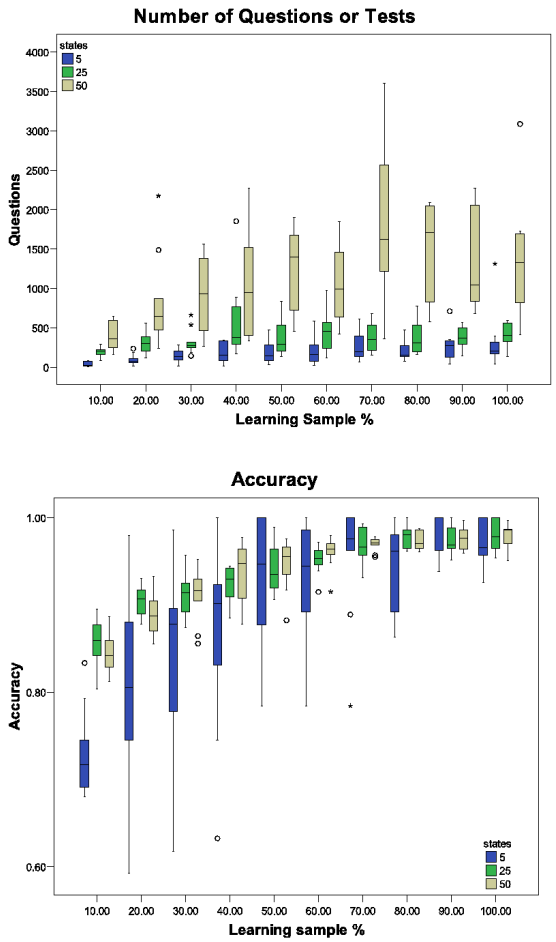


Figure 7. Results for 5, 25 and 50 states, where the complete training set contains $4n$ strings

4.2. Results and Discussion

The purpose of these results is to provide an insight into the scalability and accuracy of the technique. The standard process for evaluating state-merging grammar inference techniques, which is inspired by Lang *et al's* Abbadingo competition [16], is to generate random machines of various sizes, along with a sample of random walks across the machine. An equal number of accepted and rejected walks are used to infer the hypothesis machine, and a further sample of random walks in the target machine are used to establish the accuracy of the hypothesis machine.

The performance study presented here differs slightly from previous grammar inference evaluations. The technique presented here presumes that, at least to begin with, the algorithm will only be supplied with positive traces of the software system. Thus, whereas previous studies supply an equal number of accepted and rejected sequences, our study assumes that only positive sequences are supplied, and relies on the question generation process to produce a sufficient set of negative questions. Three sets of ten random graphs of sizes n =5, 25 and 50 states were generated, with an alphabet of six transition functions (previous similar evaluations have only used an alphabet of two). The graphs are deliberately densely connected, with many state transitions labelled by multiple functions, in an attempt to synthesise state machines of relatively complex software systems.

A training set of size $4n$ with distinct random sequences was generated, where the length of a sequence forms a roughly uniform distribution $[0, p + 5]$ where p is the depth of the machine. The test set, which is used to measure the accuracy of the hypothesis machine, was created in a systematic manner by using the W-Method [4] state machine testing technique (also ensuring that there was no overlap between the test and training samples). For each random graph, the inference algorithm was applied to 10%, 20% ... 100% of the $4n$ samples. Each hypothesis was evaluated in terms of (a) the number of questions that were asked to reach the final solution and (b) the accuracy with respect to the set of test sequences. Due to space restrictions we have not included the exhaustive set of results, but try to quote relevant results when appropriate.

Bearing in mind the complexity of the synthesised machines (alphabet of size 6, large numbers of multi-function transitions) and the fact that we start off with only positive samples, these preliminary results are encouraging. They show that the technique is capable of producing relatively accurate results from a sparse sample of initial sequences (i.e. dynamic traces). The results for 5 states at 10% rarely achieve an accuracy of above 80%, but this is primarily because they are only given two strings to start with. As the number of initial strings increases, the accuracy improves. Given all $4n$ input sequences the accuracy of the technique is consistently above 95%.

Depending on the complexity of the target machine and the initial sample of data, the technique can however produce a large number of questions, which are vital to ensure that the inference step is correct. If answered automatically (by posing questions as tests to the system), this is reasonable, particularly if the transition functions are governed by simple inputs that do not imply complex data constraints on the state of the system. However, if this is not the case and they require manual intervention, the technique could be rendered impractical for complex systems. There does however seem to be much scope for reducing the number of questions. The merging heuristic is very accurate; most of the questions *confirmed* that the hypothesis machine was correct, and only around 10 - 15% of the questions ever caused the algorithm to restart. Investigating the reduction of the number of questions produced is an important part of

our future work.

5. Related Work

A number of approaches exist that use some form of source code analysis to reverse engineer state machines, some at object-level, others at higher levels of abstraction. Previous work by the authors [22] has used symbolic execution as a basis for identifying state transitions between manually supplied abstract states. Authors including Duarte *et al.* [7] and Whaley *et al.* [23] also use source code as a basis for reverse engineering state machines, but augment their analysis with dynamic traces of the system. These approaches differ from the approach presented in this paper in two ways. Firstly, they all require some form of analysis of the underlying source code structure in terms of its data / control flow. Our approach may require the user to inspect the source code to a more superficial extent in order to identify which method sequences abstract to the higher-level functions that label the state transitions. Secondly our approach is interactive and, if questions are interpreted as tests, suggests which sequences of inputs need to be executed in order to produce an accurate result. This is particularly beneficial, because it lessens the risk of producing an incomplete result, a problem that is intrinsic to most techniques that are based entirely on dynamic analysis.

The idea of constructing a state machine of a program from example computations was conceived by Biermann and Feldman [2]. They proposed the k-tails algorithm which, like the QSM algorithm, merges states based on the similarity of their behaviour. A number of other authors (e.g. [5, 21, 19]) have used variants of such algorithms to generate state machines from scenarios (usually manually generated scenarios). One weakness of these algorithms is that they do not incorporate membership queries and, in the case of k-tails at least, only work from positive samples. Generalising a set of samples to a state machine if only positive samples are provided is reasonable under the condition that the sample is complete (i.e. every possible sequence up to a given length is provided). Dynamic analysis is however generally prone to incompleteness which is why our application demands the presumption that the provided sample is not complete, in which case an interactive approach that accepts negative sequences is more appropriate.

As mentioned in the introduction, Hungar *et al.* [13] address this problem by applying Angluin's L* inference technique [1] to reverse engineering the state machine structure of a phone system. There are two key differences between the L* technique and the QSM approach adopted in this paper. Both assume that the sets of input sequences are incomplete, but the L* technique proceeds to systematically and comprehensively explore the state space of the target machine, which can become extremely expensive if the underlying state machine is complex. The QSM technique on the other hand presumes that the input sequences at least offer some basic coverage of the essential functionality of the system (they are characteristic of the underlying transition structure), in which case the machine can be inferred relatively cheaply by a process of state merging.

Our technique depends on the developer to provide a set of abstraction functions that map from some pattern of method sequences that occur in the trace to an abstract function that represents a user-level system function. Although this has been relatively straightforward in our experience with the case study, it inevitably requires some degree of familiarity with the underlying system. A number of feature-identification techniques do exist that can facilitate this. An approach by Eisenbarth *et al.* [9], which is particularly suitable with the availability of dynamic traces as produced by our technique, combines formal concept analysis and trace analysis to similarly map user-level functions to their implementation.

6. Conclusions and Future Work

This paper has presented the application of an interactive, tractable grammar inference approach to the problem of reverse engineering state machines from software implementations. The technique generates hypothesis machines from sequences of valid system functions. As new hypothesis machines are produced, queries are presented in the user, which can either be answered manually, or fed to the system as tests. The integrated question / test generation technique aims to ultimately produce a set of sequences that are a characteristic sample of the underlying state machine which, if true, will result in a complete, minimal and deterministic state machine of the underlying system. This ultimately allows the application of powerful functional testing techniques, and also serves as a useful means to comprehensively document system behaviour.

The implementation of the technique has enabled the generation of some preliminary data. These show that, even with a sparse set of initial sequences, the technique will still produce a reasonably accurate machine. Given that the final machine is subject to a manual inspection, it can be used as a reliable basis for the application of established model-based testing techniques [17], which can be used to comprehensively establish whether the underlying system conforms to the model and is therefore correct.

The technique can, depending on the initial sequences and complexity of the underlying machine, produce a substantial number of questions. If these can be answered automatically as system tests, this is not a problem. If on the other hand they have to be answered manually, it could render the technique impractical. However, it can also be argued that the necessity of a large number of execution se-

quences is essential to any dynamic analysis technique if its results are to be reasonably sound, in which case this technique excels because it provides a powerful guidance mechanism. We also believe that there is scope for a substantial reduction in the number of questions asked; if only 10-15% of them result in a restart, and most of them confirm the hypothesis, a major proportion of them are possibly unnecessary.

In the immediate future we will work on ways to reduce the number of questions asked. This will involve looking at the distribution of confidence scores associated with the questions, and investigating the use of score thresholds to keep the number of questions to an effective minimum. At the same time we will take a more applied approach in the evaluation of this technique, by evaluating it in a more controlled manner, with respect to larger scale real-life systems (as well as synthesised ones).

Acknowledgments We thank Pierre Dupont for helping us to gain a better understanding of his QSM approach and providing a number of insights that helped in its implementation. Our work is supported by EPSRC grant EP/C51183/1.

References

[1] D. Angluin. learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[2] A. W. Biermann and J. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 21:592–597, 1972.

[3] E. Börger. Abstract state machines and high-level system design and analysis. *Theoretical Computer Science*, 336(2-3):205–207, 2005.

[4] T. Chow. Testing Software Design Modelled by Finite State Machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

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

[6] C. Damas, B. Lambeau, P. Dupont, and A. van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering*, 31(12):1056–1073, 2005.

[7] L. M. Duarte, J. Kramer, and S. Uchitel. Model extraction using context information. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 380–394. Springer, 2006.

[8] P. Dupont, B. Lambeau, C. Damas, and A. van Lamsweerde. The QSM algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*, 2007. to appear.

[9] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[10] M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[11] M. Holcombe and F. Ipate. *Correct Systems - Building A Business Process Solution*. Applied Computing Series. Springer, 1998.

[12] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[13] H. Hungar, T. Margaria, and B. Steffen. Test-based model generation for legacy systems. In *ITC*, pages 971–980. IEEE Computer Society, 2003.

[14] H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *International Conference on Computer Aided Verification (CAV'03)*, 2003.

[15] K. Lang. Random DFA's can be approximately learned from sparse uniform examples. In *COLT*, pages 45–52, 1992.

[16] K. Lang, B. Pearlmutter, and R. Price. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In V. Honavar and G. Slutzki, editors, *Grammatical Inference; 4th International Colloquium, ICGI-98*, volume 1433 of *LNCS/LNAI*, pages 1–12. Springer, 1998.

[17] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[18] K. Li, R. Groz, and M. Shahbaz. Integration testing of components guided by incremental state machine learning. In P. McMinn, editor, *TAIC PART*, pages 59–70. IEEE Computer Society, 2006.

[19] D. Lorenzoli, L. Mariani, and M. Pezze. Inferring state-based behavior models. In *Proceedings of the International Workshop on Dynamic Analysis (WODA'06)*, 2006.

[20] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In N. P. de la Blanca, A. Sanfeliu, and E. Vidal, editors, *Pattern Recognition and Image Analysis*, volume 1 of *Series in Machine Perception and Artificial Intelligence*, pages 49–61. World Scientific, Singapore, 1992.

[21] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, and F. I. Vokolos. Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In *ICSM*, pages 155–164. IEEE Computer Society, 2005.

[22] N. Walkinshaw, K. Bogdanov, and M. Holcombe. Identifying state transitions and their functions in source code. In *Testing: Academic and Industrial Conference (TAIC PART'06)*, pages 49–58. IEEE Computer Society, 2006.

[23] J. Whaley, M. Martin, and M. Lam. Automatic Extraction of Object-Oriented Component Interfaces. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2002.