Automated Comparison of State-Based Software Models in terms of their Language and Structure

Neil Walkinshaw
Department of Computer Science, The University of Leicester
and
Kirill Bogdanov
Department of Computer Science, The University of Sheffield

State machines capture the sequential behaviour of software systems. Their intuitive visual notation, along with a range of powerful verification and testing techniques render them an important part of the model-driven software engineering process. There are several situations that require the ability to identify and quantify the differences between two state machines (e.g. to evaluate the accuracy of state machine inference techniques is measured by the similarity of a reverse-engineered model to its reference model). State machines can be compared from two complementary perspectives: (1) In terms of their *language* – the externally observable sequences of events that are permitted or not, and (2) in terms of their *structure* – the actual states and transitions that govern the behaviour. This paper describes two techniques to compare models in terms of these two perspectives. It shows how the difference can be quantified and measured by adapting existing binary classification performance measures for the purpose. The approaches have been implemented by the authors, and the implementation is openly available. Feasibility is demonstrated via a case study to compare two real state machine inference approaches. Scalability and accuracy are assessed experimentally with respect to a large collection of randomly synthesised models.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*State diagrams*

General Terms: Documentation, Algorithms, Measurement

Additional Key Words and Phrases: Labelled Transition Systems, Accuracy, Comparison

This paper is an extension of two papers presented at ICGI 2008 [Walkinshaw et al. 2008] and WCRE 2009 [Bogdanov and Walkinshaw 2009]. The authors are supported by the EPSRC REGI grant (EP/F065825/1) and the EPSRC STAMINA grant (EP/H002456/2).
The Authors' addresses: Neil Walkinshaw, Department of Computer Science, Leicester University, University Road, Leicester, LE1 7RH.; Kirill Bogdanov, Department of Computer Science, Sheffield University, Regent Court, 211 Portobello Street, Sheffield, S1 4DP.
email:nw91@le.ac.uk, k.bogdanov@dcs.shef.ac.uk

1. INTRODUCTION

Software behaviour, whether at an object, component or system-level is often modelled in terms of state machines. Formalisms such as StateCharts [Harel and Naamad 1996], Labelled Transition Systems, Abstract State Machines [Börger 2005], X-Machines [Laycock 1992] and Extended Finite State Machines [Cheng and Krishnakumar 1993] have been developed to model software at various levels of abstraction in terms of its sequential, state-based behaviour. With the increasing popularity of model-driven software development, the use of such models has become widespread in the software development process.

State machine models are valuable for a range of development and maintenance tasks. They help the developer to understand how the system works; they have an intuitive diagrammatical notation, which graphically captures its sequential behaviour. By documenting the system behaviour in a formal way, they also form the basis for a number of automated verification techniques such as model-checking [Clarke et al. 1999], and model-based testing techniques [Lee and Yannakakis 1996].

The ability to compare different state machines is important for a range of software-engineering scenarios. For example, the developer might have models of two different versions of the same software system, and need to quantify or understand the differences between the two. Alternatively, the two models might be from two different software systems that attempt to solve the same problem. In another scenario (the main motivation for the authors), the task might be to compare a reverse-engineered state machine to some ideal model that has been generated by hand, for the sake of evaluating the accuracy of the reverse-engineering technique. Again, a suitable approach is required to capture any inaccuracies in the model, and to point out where it has been successful.

As mentioned previously, there are a range of specific state machine formalisms that include their own notions and corresponding notations. For example, StateCharts can model hierarchical states and most formalisms have their own means of linking data guards or transformations to specific transitions. This work adopts the simplest possible interpretation of a state machine (yet one that can ultimately be used to represent most state-based models): a labelled transition system, where each transition is labelled by a symbol.

Using this notation, state machines can be interpreted from two perspectives. From the *language* perspective, a state machine is interpreted as a set of possible or impossible sequences of events (transition labels); these represent the externally visible behaviour of a state machine. From the alternative *structural* perspective, a state machine can be interpreted in terms of the states and transitions that constitute its transition structure. These two perspectives are complementary; two machines may have a reasonably similar state transition structure, but completely different languages and vice-versa. Consequently, in order to obtain a complete picture of how two state machines relate to each other, it is important to consider both perspectives.

Several approaches exist to compare state machines. Existing language comparison approaches fail to effectively and reliably compare the full languages because they do not consider the essential impossible sequences alongside the possible ones. Existing structural comparison approaches can also be unreliable; they either rely

on equivalent states having the same state labels or assume that equivalent states (with the same subsequent behaviour) can be reached by the same path from the initial state, which is not necessarily the case.

This paper presents two techniques that address these problems. The first technique compares their languages, and the second compares their structures. Both techniques are associated with a single measurement – the F-Measure – that enables the differences between two models to be precisely quantified.

Section 2 contains the definitions of labelled transition systems (which are used to represent state machines), along with their languages. It motivates the need to compare state machines, describes why this is difficult, and shows how existing approaches attempt to solve this problem. Section 3 introduces our technique to compare the languages of two state machines. Section 4 presents our technique to compare their structures. Section 5 presents a small case study, that shows how the two techniques can be used to comparatively evaluate the accuracy of two state-machine reverse-engineering techniques. Section 6 presents an empirical evaluation that studies the accuracy and performance of the presented techniques against a large set of artificial transition systems that are generated to resemble software models. Section 7 presents related work, and finally section 8 presents our conclusions and future work.

2. BACKGROUND

This section provides a brief introduction to the notation that will be used throughout the paper to represent state machines and their languages. This is followed by an introduction to the problem of comparing state machines. It will provide a brief overview of existing approaches (which will be elaborated in the 'Related Work' section), and will motivate the need for improved and more authoritative comparison techniques.

2.1 Notation: State Machines and Languages

We represent a state machine as a Labelled Transition System (LTS), which reduces a state machine to a set of states and transitions, where each transition is labelled by a symbol or string.

Definition 2.1 Labelled Transition System (LTS). *An LTS is a quadruple* (Q, Σ, Δ, q_0)*, where* Q *is a finite set of states,* Σ *is a finite alphabet,* $\Delta \subseteq Q \times \Sigma \times Q$ *is the transition relation and* $q_0 \in Q$ *is the initial state. This can be visualised as a directed graph, where states are the nodes, and transitions are the edges between them, labelled by their respective alphabet elements. An LTS is* deterministic *if every state has at most one outgoing transition for a given label, i.e. the cardinality of the set of target states in the transition function is always 1. An LTS is* minimal *if no two states are equivalent (i.e. accept the same language - as defined below).*

When referring to multiple LTSs, a subscript is used to indicate the LTS being referred to. So given an LTSs X, Q_X refers to the set of states in X. The labelled arrow $p \xrightarrow{\sigma} q$ denotes that there is a transition from state p to state q, labelled by σ. $p \xrightarrow{\sigma}$ denotes that there exists an outgoing transition from state p that is labelled by σ, where it does not matter what the target state is. Similarly, $\xrightarrow{\sigma} q$ denotes

that there exists an incoming transition to state q that is labelled by σ, where it does not matter what the source state is. The *outgoing alphabet* for some state a in a machine X is defined as $\Sigma_X^{out}(a) = \{\sigma \in \Sigma_X | a \xrightarrow{\sigma}\}$. Similarly the *incoming alphabet* for some state b in a machine Y is defined as $\Sigma_Y^{inc}(b) = \{\sigma \in \Sigma_Y | \xrightarrow{\sigma} b\}$.

To define the language of an LTS, we draw on the inductive definition for an extended transition function $\hat{\delta}$ used by Hopcroft *et al.* [Hopcroft et al. 2007].

Definition 2.2 The Language and its Complement. *For a state p and a string w, the extended transition function $\hat{\delta}$ returns the state q that is reached when starting in state p and processing sequence w. For the base case $\hat{\delta}(p, \epsilon) = p$. For the inductive case, let w be of the form xa, where a is the last element, and x is the prefix. Then $\hat{\delta}(p, w) = \delta(\hat{\delta}(p, x), a)$.*

Given some LTS A, for a given state $q \in Q$, $L(A, q)$ is the language of A in state q, and can be defined as: $L(A, q) = \{w | \hat{\delta}$ is defined for $(q, w)\}$. The language of an LTS A can be defined as $L(A) = \{w | \hat{\delta}$ is defined for $(q_0, w)\}$. This will be necessary to define the test set in section 3.1.

The complement of a language L for a machine A (i.e., the set of sequences that do not belong to $L(A)$) is denoted by $L(A)^C$.

This paper deals with the problem of comparing two LTSs to each other. It adopts the convention that one of them represents the reference to which the other is being compared. The terms "reference", "target" or "correct" model are used interchangeably. The other machine will usually be referred to as the "subject" machine. The terms "state machine" and "labelled transition system" or "transition system" are also used interchangeably.

2.2 Motivation

The task of comparing state machines to each other is well-established. Verification tasks such as model-checking and model-based testing routinely compare specification state machines to state machines representing the implementation. The aim is usually to establish whether a property holds or not – whether the implementation conforms to the specification.

In this paper, the goal is somewhat different. It is not to establish a boolean truth about whether a machine does or does not conform to another, but is instead to enable a more quantitative, detailed comparison of the two models. Instead of stating whether or not one model is equivalent to another, we want to quantify this equivalence, and to provide the means to determine in what sense two models overlap, and in what sense they differ.

The techniques developed in this paper are primarily motivated by the specific challenge of evaluating the accuracy of state-machine reverse-engineering techniques. Given a software system, along with an ideal target model that was generated manually, the task is to evaluate the accuracy of the reverse-engineered model with respect to the target model. There are however several other tasks for which such methods would be of value. In a model-driven development environment, the task of analysing the differences between different models can arise in several contexts (e.g. the comparison of different proposed specifications, or the difference between different versions of the same system). In the domain of software testing, techniques are being developed to test black-box systems without specifications by

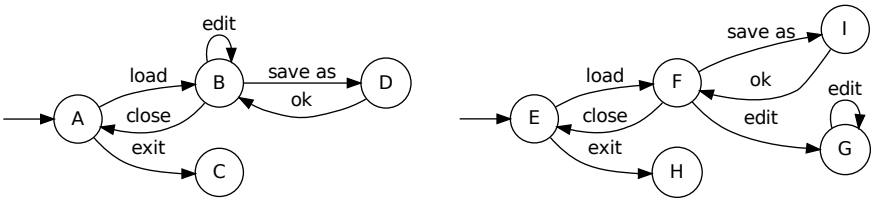


Fig. 1. A state machine for a fictional text editor, and a slightly different version

inferring models from test sets [Weyuker 1983; Walkinshaw et al. 2009], thus the ability to compare these models would in turn enable the comparison of test sets in terms of the extent to which they exercise a system.

Figure 1 illustrates the problem of state-machine comparison. The machine on the left represents the LTS for a simple fictional text editor (the reference machine). Let us suppose that the machine on the right has been reverse-engineered (the subject machine). How do we quantify the accuracy of the subject machine? How do we characterise the differences between them?

There are two perspectives from which to answer these questions. From the *languages* perspective it is possible to compare the sequences of labels that are possible or impossible in the two machines, and establish the extent to which these overlap. From the *structural* perspective, it is possible to investigate the extent to which their states and transitions overlap.

It is worth noting that, for the work in this paper, it is presumed that a given event is given the same label in both LTS models. Depending on the application scenario this will not necessarily be the case. The specific problem of label matching is considered beyond the scope of this paper, but can be addressed by using established NLP label matching algorithms (as proposed by Nejati *et al.* in their work on state machine comparison [Nejati et al. 2007]). The rest of this section provides a more detailed introduction to the two notions of comparing an LTS in terms of its language and structure.

2.2.1 *The Challenge of Comparing LTS Languages.* In comparing the languages of two machines, we are comparing two sets of sentences (where a sentence is a path through the machine, denoted as a sequence of symbols used to label transitions on the path). Such a comparison is trivial if the two sets are finite. The challenge in comparing the languages of LTSs is due to the fact that their structures tend to be cyclic, which means that they can produce an infinite number of different sentences. It is impossible to exhaustively enumerate every sentence in the language of either machine.

The most common approach [Lang et al. 1998] is to take a random sample from the language of the reference machine (assuming that this in some sense "characterises" the language as a whole), and then to count the proportion of these that are correctly classified (accepted or rejected) by the subject machine. The sample is usually split, ensuring that half of the sample belongs to the language of the reference LTS, and the other half belongs to its complement. The higher the proportion of correctly classified samples, the more likely the two LTSs are to be equivalent.

Two important problems arise with this approach. The first problem is that the validity of the comparison is questionable, because it is virtually impossible to obtain a truly representative sample of the language of a machine by random selection. If the sample is computed by random walks across the transition structure, it will inevitably be biased towards those parts of the target machine that are easiest to reach, and emphasise a selective portion of the language. An alternative approach is to simply produce random strings from the alphabet alone (i.e. to generate sequences in Σ^*), and to then classify these as belonging to the language or not with respect to the reference LTS. The problem here is that software models tend to be structured in such a way that only a small fraction of random sequences belong to their language, making the likelihood of randomly obtaining a fully representative sample extremely remote. The challenge is exacerbated for larger reference machines. Experiments by Lang [Lang 1992] in the domain of regular grammar inference suggest that the size of a random sample has to grow by orders of magnitude in relation to the state machine if it is to even approximately represent its behaviour.

The second problem is that the single-value metric (the proportion of sequences in the sample that are are classified equivalently by both machines) is not particularly useful. It provides little in the way of qualitative insights about differences between the two LTSs. For example, if the result of a comparison is 50% accuracy, what does this mean? Is the subject machine more general (does its language accept more samples than the reference LTS), or is it more specific (does it reject more samples than the reference machine)?

A suitable approach to compare two languages will need to address these two problems. It will have to systematically identify a complete set of samples from the target machine that can be used to compute an authoritative comparison. It will also need to do so with a metric that provides more insights about why the languages are (dis-)similar.

2.2.2 *The Challenge of Comparing LTS Structures.* Language comparisons only provide a limited insight into the (dis-)similarities of two LTSs. A comparison of their state transition structures can often provide complementary insights that could not be obtained by merely treating them as black-boxes. Currently there is only a limited amount of work that compares state machines in terms of their structures [Nejati et al. 2007; Quante and Koschke 2007; Kelter and Schmidt 2008; Pradel et al. 2010].

Structurally comparing two non-trivial LTSs is difficult. The task essentially involves establishing which states and transitions in both machines appear to be equivalent, and then working out which states and transitions must have been added or removed in the subject LTS. A number of structural comparison techniques exist for richer structures such as StateCharts [Nejati et al. 2007; Kelter and Schmidt 2008], which take advantage of additional information contained in the representation of states and transitions. LTS specifications are somewhat simpler, and arguably pose a harder problem with respect to comparison. States are unlabelled, and represent mere points in time that are used to specify a partial order of events as specified by the state transition labels. Accordingly, one cannot rely on state-annotations. Any two states in the two machines could potentially

be equivalent. Whether or not this is actually the case can only be established by pairing up the surrounding states and transitions – a process that can become very expensive. The challenge lies in doing so in an efficient manner.

3. COMPARING LANGUAGES

This section presents an approach to comparing two LTSs in terms of their permitted sequences of events, i.e. their languages. It shows how techniques from the domain of model-based testing can be applied to sampling the language of the target state machine in such a way that it is systematically "covered" [Walkinshaw et al. 2008]. It also shows how generated test sets can be used to provide a more detailed measure of how the languages of the two LTSs overlap.

3.1 Generating a Suitable Finite Sample of the Language of an LTS with Model-Based Testing Techniques

As stated in section 2.2.1, it is impossible to explicitly compare the languages of two LTSs if they are infinite. It is highly unlikely that random samples will adequately represent the underlying languages. Instead of random sampling, this section shows how existing techniques from the field of model-based testing can be used to systematically traverse an LTS structure to compute a finite sample that is representative of its language (provided that the LTS in question obeys certain basic properties).

In a model-based testing scenario [Lee and Yannakakis 1996], the assumption is that we have a model, and that the system we are attempting to test is hidden – we can only observe whether or not a sequence of inputs causes it to produce the expected outputs. The challenge is to ensure that inputs to the implementation lead it into the correct states, where the subsequent range of inputs and outputs is exactly the same as in the model. A model-based test set generator generates tests from the model in order to establish whether this is the case. If an implementation produces a different output to the model in response to any of these test sequences, the implementation is considered to be faulty. Otherwise the developer can be reasonably confident that the implementation in question is correct. Different testing methods can establish the similarity of a model and its implementation under varying assumptions (see Lee and Yannakakis for a more comprehensive overview of the area [Lee and Yannakakis 1996]).

The challenge of establishing whether some hidden implementation conforms to an LTS model (i.e. to establish that the implementation and model LTSs are identical) cannot be solved by testing in the most general case. It is always possible for the hidden implementation to contain extra, potentially faulty states than remain unexplored by the test set. For this reason, testing techniques assume that it is possible to estimate the maximal number of states in the implementation, so that these extra states can be accounted for in the test set (though this involves increasing its size exponentially).

If the maximal number of states in the implementation is known[1], and the assumption can be made that both the model and implementation LTSs are deter-

[1]In the traditional testing context, it is usually impossible to know the maximum number of states, and this has to be estimated.

ministic and minimal, it becomes possible to generate a finite set of tests that can be used to guarantee that the languages of two machines are equivalent. A number of techniques exist to compute these test sets [Lee and Yannakakis 1996]. The testing method used in this paper is the Vasilevski/Chow W-Method [Vasilevskii 1973; Chow 1978].

To provide an informal overview of the method, the aim of the test set is:(1) to reach every state, (2) to ensure that there are no unexpected outgoing transitions from these states, and (3) to make sure that the correct state has been reached (the state is characterised in terms of the subsequent sequences of symbols that can be taken from it). The test set is effectively the cross product of three sets of sequences: A "state cover" – sequences that lead to each state, a set of symbols in Σ, and a "characterisation set", a set of sequences that have been computed to distinguish the outgoing behaviour of every pair of states (thus ensuring that the current state is the correct one).

The following more formal description of the W-Method follows a description by Bogdanov *et al.* [Bogdanov et al. 2006]. Given some implementation (interpreted to be a hidden LTS S), and a specification LTS A, the W-Method constructs a set of sequences that should be classified equally by both LTSs. The set of sequences $Y \subseteq \Sigma^*$ is constructed to ensure that $(L(A) \cap Y = L(S) \cap Y) \Rightarrow L(A) = L(S)$. Such a set is called a *test set* of A.

Definition 3.1 State Cover. *A* state cover C *is a prefixed-closed set containing all sequences of inputs needed to visit every state of an LTS A from the initial state. That is, C is a subset of $L(A)$ such that $\epsilon \in C$ and for all states $q \in Q \setminus \{q_0\}$ there exists a $c \in C$ such that $\hat{\delta}(q_0, c) = q$.*

Definition 3.2 W-distinguishability, the Characterisation set. *For a subset $W \subseteq \Sigma^*$ the states $q_1, q_2 \in Q$ are called* W-distinguishable *if $(L(A, q_1) \cap W) \neq (L(A, q_2) \cap W)$. We call W a* characterisation set *[Gill 1962] of A if any two distinct states of A are W-distinguishable.*

Besides the LTS A from which the test set is constructed, the $W-$method is parameterised by positive integer k. If the implementation has additional erroneous states, their detection can only be guaranteed if their potential presence is explicitly accounted for. The k parameter serves as a conservative estimate of the potential number of extra states. Once k has been selected, the test set Y is computed by the cross-product of the different sets of sequences: $Y = C(\{\epsilon\} \cup \Sigma \cup \cdots \cup \Sigma^{k+1})W$.

The sequences in C ensure that the test set reaches every state. The appended sub-sequences consisting of permutations of Σ up to a length of k try every combination of possible inputs from each state, in an attempt to enter unexpected states in the implementation. The W set makes sure that the state reached by the preceding combination of inputs is the intended one. As a whole, the test set is guaranteed to establish whether there are any differences in the languages of the two machines (assuming of course that k is an upper bound on the number of extra states in the subject machine).

	Classification by Subject machine S	
Classification by Reference machine R	$seq \in L(S)$	$seq \in L(S)^C$
$seq \in L(R)$	True Positive (TP)	False Negative (FN)
$seq \in L(R)^C$	False Positive (FP)	True Negative (TN)

Table I. Confusion matrix for classification of sequence seq. A sequence is positive if it is accepted by the LTS, and negative otherwise.

3.2 Using Test Sets to Compare LTS Languages

As discussed in the previous section, comparing the languages of two LTSs is particularly challenging; to do so in an authoritative way requires the two languages (i.e. two sets of sequences) to be represented in a finite way. The traditional approach, of simply taking a random sample to represent the language of a LTS is unreliable, because it is difficult to obtain a sample that is truly representative of its language. Test-generation techniques offer a more systematic, rigorous approach of exploring the behaviour of a LTS (with a finite set of sequences), and can therefore provide a more authoritative basis for comparing the languages of two machines. Importantly, the issue of guessing a suitable value for k that hampers the W-Method in its traditional software testing context is no longer an issue here, because both machines are visible to us. The rest of this section shows how the languages of two machines can be compared, using the W-method as a basis for representing the language of a LTS.

The W-Method makes it possible to systematically generate a finite set of sequences that are representative of a given LTS. The languages of two LTSs X and Y can be compared by simply generating the test set from X, and measuring the proportion of test sequences classified the same way by X and Y. Given that neither machine is hidden, it is possible to work out k by subtracting the number of states in Y from the number of states in X (using $k = 0$ if the Y has fewer states). In other words, it is possible to generate a test set that is guaranteed to comprehensively exercise every state, ensuring that the origins of any difference in the languages of the two machines will be exposed.

There still remains the challenge of measuring the differences between two languages. As stated previously, merely using a single measure (i.e. proportion classified the same way) can be uninformative and even misleading, especially when the vast majority of tests produced by the W-Method should be rejected. The following subsection shows how established binary classification performance measures can be adopted to produce a more reliable measure.

3.2.1 *Using Binary Classifier Performance Measures to Compare LTS Languages.* To compare the similarity of two languages, the test cases are categorised in terms of a *confusion matrix* [Sokolova and Lapalme 2009] (a table that separates out true and false positives and negatives). This is shown in Table I; the set of *true positives* refers to the set of sequences that are accepted by both R and S, *false positive* refers to the set of sequences that are accepted by S but not by R and so on. Partitioning the languages in this way has two main benefits. Firstly, we no longer need to ensure that the test set is balanced evenly between accepting and re-

Measure	Formula	Interpretation
Precision	$\frac{\|TP\|}{\|TP \cup FP\|}$	Proportion of tests in $L(S)$ that are in $L(R)$
Recall (Sensitivity)	$\frac{\|TP\|}{\|TP \cup FN\|}$	Proportion of tests in $L(R)$ that are in $L(S)$
F-Measure	$\frac{2 * Precision * Recall}{Precision + Recall}$	Harmonic Mean between Precision and Recall
Specificity	$\frac{\|TN\|}{\|TN \cup FP\|}$	Proportion of tests in $L(R)^C$ that are in $L(S)^C$
Balanced Classification Rate (BCR)	$\frac{Sensitivity + Specificity}{2}$	Accuracy of $L(S)$, balanced with respect to positive and negative classification

Table II. Performance Measurements for Comparing LTS Languages

jecting test-cases. This enables the use of techniques such as the W-Method, which is inherently imbalanced because (for typical machines) it contains a large majority of test cases that should be rejected. Furthermore, it enables the use of a range of well-established performance measures that are computed in terms of these four quantities. Measurements such as Precision, Recall, BCR and F-Measure [Rijsbergen 1979; Sokolova and Lapalme 2009], which are well established in Information Retrieval, can now be applied to measure the accuracy of the language of an LTS.

Table II lists the key performance measurements, and shows how they would be interpreted with respect to LTSs. The choice of which metric to use ultimately depends on the context in which the two LTSs are being compared. If we want to ensure that the subject machine captures the language of the reference machine, and do not care so much about ensuring accuracy with respect to language complements, we can concentrate on Precision and Recall (using the F-Measure to aggregate the two). If, however, we want to place an equal emphasis on the accuracy with respect to the language and its complement, it makes more sense to use Sensitivity and Specificity, and to aggregate them with the BCR measure.

3.3 Example: Comparing the Languages of two State Machines of a Text Editor

The remainder of this section presents a small example of how the above measures are applied. We use the fictional text editor shown in Figure 1 as an example. To begin with, we compute the test set from the correct model. We know that the subject machine has one additional state to the correct one, so we set the k value for computing the test set to 1. This results in a set of 52 sequences; the relatively high number is explained by the need to attempt every element in the alphabet (six symbols) from each of the four states. The test sequences, along with their classifications by the reference and subject LTSs are provided in appendix A.

The confusion matrix that results from a comparison of the classifications for the test set by the subject and reference LTSs is shown in Table III. The various measurements that are derived from this table are displayed in Table IV. The perfect precision score tells us that all of the sequences that are accepted by the subject machine are also accepted by the reference machine (there are no false

	Subject machine S	
Reference machine R	in $L(S)$	in $L(S)^C$
in $L(R)$	$\|TP\| = 3$	$\|FN\| = 4$
in $L(R)^C$	$\|FP\| = 0$	$\|TN\| = 45$

Table III. Confusion matrix for small text-editor example

Measure	Score
Precision	1
Recall (Sensitivity)	0.43
F-Measure	0.60
Specificity	1
Balanced Classification Rate (BCR)	0.71

Table IV. Scores derived from confusion matrix in Table III

positives). However, the Recall score is relatively low (0.43), which tells us that the language of the subject LTS is missing a substantial proportion of the sentences that should be present according to the reference LTS. These two coordinates are summarised by the F-Measure. The specificity score tells us that the accuracy of the subject machine with respect to the language complement is perfect - all of the tests that should be classified as negative are actually classified as negative (there are no false positives). The BCR score of 0.71 averages the performance at classifying positive and negatives (sensitivity and specificity). Altogether, these measurements tell us that the subject machine is accurate with respect to the complement of its language (it is good at rejecting the tests it is supposed to reject), but inaccurate with respect to its language. Although Precision is high, it rejects too many tests that it is supposed to accept, which explains the low recall.

From Table III, the unbalanced nature of the test set produced by the W-Method is evident. The vast majority of the tests (45) are negative, whereas only 7 are positive. This balance between positive and negative sequences ultimately depends on the topology of the reference machine. A smaller alphabet, or a denser transition structure (where there are more outgoing transitions from each state) would mean that a larger proportion of tests are accepted. Different testing techniques would produce test sets in a different fashion, with a different balance between negative and positive tests. Although the W-Method certainly provides a more reliable sampling basis than random tests, there are alternative test generation strategies. Our future work (discussed in more detail in section 8) will investigate the use of alternative model-based testing strategies, and the effect that these have on the final measurements.

4. COMPARING TRANSITION STRUCTURES

A complementary approach to comparing LTSs in terms of their languages is to do so in terms of their states and transition structures. This presents an alternative

perspective on LTS similarity by accounting for the internal structure as opposed to the externally observable sequences of labels. In this section we present the *LTSDiff* algorithm[2]. Like the traditional `diff` algorithm for text files, it computes the precise difference between two LTSs, returning the missing or superfluous states and transitions.

The language-based approach presented previously works in terms of a test set, where every test is a sequence of events that start from the initial state. The difference between two machines is established in terms of the different sequences of labels that are possible from the intial state. However, a minor difference in language can mask a major structural difference in the LTS, and the technique presented here shows how such differences can be exposed.

Instead of treating the LTS as a language-producer, the approach presented here treats it as a structure; differences are no longer established in terms of sequences of labels, but in terms of actual transitions or states. The algorithm revolves around the computation of scores that measure the similarity of individual pairs of states. This score is computed by matching up the surrounding network of states and transitions. Computing the overlap of the surrounding behaviour from two states is a recursive process. It is first necessary to establish the overlap of their immediate surrounding transitions (local similarity), and then to consider the similarity of the target / source states of these transitions (global similarity).

Section 4.1 describes how two states can be compared in terms of the overlap of their adjacent transitions, section 4.2 then gives a recursive extension accounting for similarity of their adjacent states. The full algorithm that integrates these measures is then presented in section 4.3, and means by which to measure these differences in terms of Precision and Recall are presented in section 4.4. Finally, section 4.5 shows an example of how the technique is applied with respect to the small text editor example.

4.1 Scoring local similarity

In this subsection we describe the process of calculating local similarity of two states in an LTS. Essentially, the local similarity S_{ab} of two states a and b is computed by dividing the number of overlapping adjacent transitions by the total number of adjacent transitions. Given a set Γ, we denote the number of elements in Γ as $|\Gamma|$. Using this notation, $S_{ab} = \frac{|matchingAdjacentTransitions|}{|AllAdjacentTransitions|}$.

Assuming that the state machines are deterministic, the score can simply be computed by $S_{ab} = \frac{|\Sigma(a) \cap \Sigma(b)|}{|\Sigma(a) \cup \Sigma(b)|}$. If there are no outgoing transitions from either of the two states, the score is considered to be zero. Examples (A,B,C) in Figure 2 all show examples for deterministic states. In (A) the outgoing transitions are identical, which produces a score of 1. In (B) there are no common outgoing transitions, producing a score of 0. In (C) only one of the two outgoing transitions from state b is matched, producing a score of 0.5.

The calculation that we actually use is a slightly expanded version that accounts for non-determinism. As an example, in Figure 2(D) it is impossible to say whether

[2]The algorithm here is presented with respect to LTS structures. An alternative version of the algorithm that addresses finite state automata (which distinguish between terminal and non-terminal states) is discussed in previous work by the authors [Bogdanov and Walkinshaw 2009].

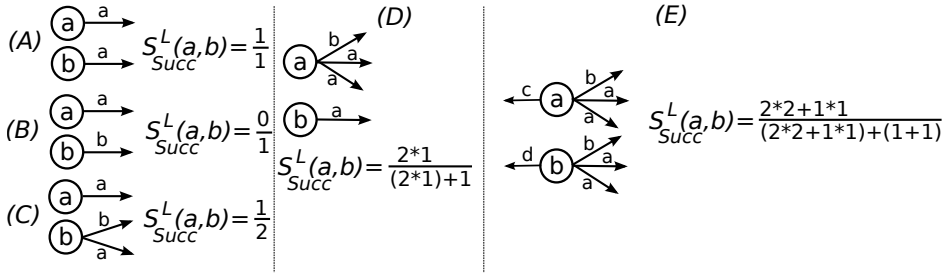


Fig. 2. Non-recursive score computation

the transition $b \xrightarrow{a}$ should be matched to the upper or lower $a \xrightarrow{a}$. In (E) there are four possible ways in which the outgoing transitions labelled a from states a and b can match each other. To account for this we take the total number of pairs of transitions that overlap and divide it by the total number of outgoing transitions that constitute possible matches. Given an LTS X and a subject LTS Y, for each label $\sigma \in (\Sigma_X \cup \Sigma_Y)$, the number of matching transitions from states $a \in Q_X$, $b \in Q_Y$ is counted in terms of the number of individual pairs of target states that can be reached by matching transitions. This is defined as follows:

$$Succ_{a,b} = \{(c, d, \sigma) \in Q_X \times Q_Y \times (\Sigma_X \cup \Sigma_Y) \mid a \xrightarrow{\sigma} c \wedge b \xrightarrow{\sigma} d\} \tag{1}$$

Given the definition for $Succ_{a,b}$, we can determine the similarity score. As mentioned previously, this is computed by dividing the number of matching adjacent transitions by the total number of possible adjacent transitions. For outgoing transitions this is computed as follows (the L superscript stands for "local"):

$$S^L_{Succ}(a, b) = \frac{|Succ_{a,b}|}{|\Sigma^{out}_X(a) - \Sigma^{out}_Y(b)| + |\Sigma^{out}_X(b) - \Sigma^{out}_Y(a)| + |Succ_{a,b}|} \tag{2}$$

The equation above computes the set of all matching pairs of target states and transition labels with respect to outgoing transitions. As a reminder, the notation $\Sigma^{out}_X(a)$ refers to the outgoing alphabet for state a of machine X (see section 2). So the expression $|\Sigma^{out}_X(a) - \Sigma^{out}_Y(b)| + |\Sigma^{out}_X(b) - \Sigma^{out}_Y(a)|$ counts the number of outgoing transitions from both states that do not match each other.

State machines characterise a state both in terms of its potential past behaviour (incoming transitions) as well as its potential future behaviour (outgoing transitions). Thus we also define the set of matching incoming transitions in a similar manner:

$$Prev_{a,b} = \{(c, d, \sigma) \in Q_X \times Q_Y \times (\Sigma_X \cup \Sigma_Y) \mid c \xrightarrow{\sigma} a \wedge d \xrightarrow{\sigma} b\} \tag{3}$$

$$S^L_{Prev}(a, b) = \frac{|Prev_{a,b}|}{|\Sigma^{inc}_X(a) - \Sigma^{inc}_Y(b)| + |\Sigma^{inc}_X(b) - \Sigma^{inc}_Y(a)| + |Prev_{a,b}|} \tag{4}$$

This approach to comparing states provides us with a flexible basis for comparing LTS's. It does not matter if we have nondeterministic states where several outgoing

transitions share the same label. Similarly, it does not matter if several incoming transitions share the same label. In principle, this approach could easily be adapted to even more complex types of transition diagrams, such as those with bidirectional state transitions.

4.2 Scoring global similarity

The definitions of S^L_{Prev} and S^L_{Succ} show how states are matched in terms of their adjacent transitions. However, we want to account for the similarity of pairs of states in terms of their wider context. When comparing a pair of states, for every matching pair of adjacent transitions we want the final similarity score to incorporate the similarity of the source or target states of these transitions as well. For two matched transitions, we want to produce a higher score if the source / target states of these transitions are almost equivalent, and a lower score if they are dissimilar. We now describe an algorithm that extends the local similarity scoring scheme to compare states recursively, creating an aggregate score by accounting for the similarity of the states that are connected to the adjacent transitions.

For the sake of simplicity we initially describe the computation procedure only in terms of outgoing transitions (the process for incoming transitions is very similar). We begin with the local similarity scoring algorithm shown in equation (4). The score is entirely dependent on the number of matching adjacent transitions $|Succ_{a,b}|$. We extend this to aggregate the similarity score for every successive matched pair of transitions (here the "G" superscript stands for "Global").

$$S^{G1}_{Succ}(a,b) = \frac{1}{2}\frac{\sum_{(c,d,\sigma)\in Succ_{a,b}}\left(1 + S^{G1}_{Succ}(c,d)\right)}{|\Sigma_X(a) - \Sigma_Y(b)| + |\Sigma_X(b) - \Sigma_Y(a)| + |Succ_{a,b}|} \tag{5}$$

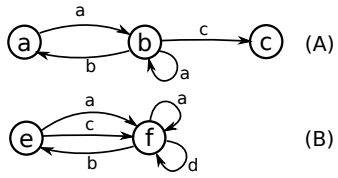
This equation augments the local similarity equation to recursively account for the similarity of next states. This can however lead to unintuitive scores because the score of every successive pair of states is given an equal precedence. The score can be skewed by high scores for state pairs that are far away. To give precedence to state pairs that are closer to the original pair of states, we introduce an *attenuation ratio* k, which gives rise to equation (6):

$$S^{G}_{Succ}(a,b) = \frac{1}{2}\frac{\sum_{(c,d,\sigma)\in Succ_{a,b}}\left(1 + k\, S^{G}_{Succ}(c,d)\right)}{|\Sigma_X(a) - \Sigma_Y(b)| + |\Sigma_X(b) - \Sigma_Y(a)| + |Succ_{a,b}|} \tag{6}$$

The fraction in front ensures that $|S^G_{Succ}(a,b)| \leq 1$. Where there are no matching transitions, $S^G_{Succ}(a,b)$ is empty, so the numerator sum is zero. In a similar way, we may define $S^G_{Prev}(a,b)$ using *Prev* instead of *Succ* and Σ^{inc} instead of Σ.

Given two LTSs X and Y, the definition $S^G_{Succ}(a,b)$ forms a system of linear equations where each variable corresponds to $S^G_{Succ}(a,b)$ for a specific pair of states $(a,b) \in Q_X \times Q_Y$. The equation system is obtained by supplying the local knowledge for every pair of states (i.e. $Succ_{a,b}$ and $\Sigma_X(a) - \Sigma_Y(b)$), so that the only unknown variables for each equation are the scores of the succeeding (or preceding) pairs of states.

Figure 3 contains an example of two state machines, and the matrix containing the system of equations that results from their comparison. The first row can be



	ae	af	be	bf	ce	cf	
ae	4			-k			1
af		6		-k			1
be			6	-k		-k	2
bf	-k			8-k			2
ce					1		0
cf						1	0

Fig. 3. Illustration of a system of equations for computation of scores in the general case.

obtained by rearranging equation (6) for (A,E). Here $Succ_{a,e} = \{(b, f, a)\}$ which means that $|Succ_{a,e}| = 1$. $|\Sigma(a) - \Sigma(e)| = 0$, $|\Sigma(e) - \Sigma(a)| = 1$. So, put together, the equation is $S(a, e) = \frac{1}{2}\frac{1+kS(b,f)}{0+1+1}$ and can be rewritten to $4S(a, e) - kS(b, f) = 1$, which provides us with the encoding for the first row in the matrix. A solution to the system of equations will produce a similarity score for every pair of states. The described systems of equations are solved separately for the forward direction and inverse, producing a solution for $S^G_{Succ}(a, b)$ and $S^G_{Prev}(a, b)$ for every pair of states. These are subsequently averaged to form a score as shown in equation (7):

$$S(a, b) = \frac{S^G_{Succ}(a, b) + S^G_{Prev}(a, b)}{2} \tag{7}$$

In theory it would be possible to assign a higher weight to either of the two directions; for instance it would be possible to add an increased weight to the forward direction to emphasise the outgoing behaviour represented by a pair of states. However, in this work we treat forward and backward directions with equal emphasis, since neither direction is of increased significance when viewing a machine from a structural perspective. When discussing this general score computation process with respect to the broader structural comparison process, it shall be denoted as $computeScores(X, Y, k)$.

Table V shows the final scores, for all of the pairs, as computed for the attenuation factor $k = 0.6$. From this table, pairs (a, e) and (b, f) produce scores that are higher than the others by a significant margin, indicating that these pairs are most likely to be equivalent. The following section will show how this scoring mechanism can be applied to compute a "diff" between two LTSs.

Pair	Forward	Backward	Average
ae	0.29	0.61	0.45
af	0.19	0	0.1
be	0.36	0	0.18
bf	0.29	0.38	0.34
ce	0	0	0
cf	0	0.13	0.07

Table V. Pair-scores computed, with k set to 0.6

4.3 The *LTSDiff* Algorithm

Our *LTSDiff* algorithm is inspired by the cognitive process a human would be expected to adopt when comparing two state machines. It is usually possible to identify 'landmarks' [Sorrows and Hirtle 1999] – certain pairs of states that can with confidence be deemed to be equivalent. Once a set of landmarks (referred to as *key pairs*) has been identified, it can be used as a basis for further comparison of the remaining states and transitions in the machines.

4.3.1 *Identifying Key Pairs (Landmarks).* Section 4.1 shows how to compute the similarity scores for two states. In our algorithm, this function shall be denoted by *computeScores*. The pairs of states with the highest scores are chosen to be *key pairs*, and are used as reference points to compute the differences between two machines by the *LTSDiff* algorithm.

Having computed the scores with the system of linear equations described in the previous section, we adopt a two-stage approach to select pairs that are most likely to be equivalent. First, we use a threshold parameter t, and consider only those pairs that fall above t (e.g. consider only the top 25%). However, a state may happen to be evenly matched to multiple other states. In practice such a one-to-many mapping is unlikely unless the structures in question have a significant amount of redundancy that produces a large degree of self-symmetry. To resolve such ambiguities, we elect to only select a pair as a key pair if its score is without doubt better than any other option. To do so, a second criterion is introduced: a ratio r. Thus, only pairs where the best match is at least r times as good as any other match are added to the set of key pairs. In our algorithm, this filtering process shall be denoted by $identifyLandmarks$, which is parameterised by the set of candidate pairs, the threshold t and ratio r.

The process of comparing two LTSs is similar to the cognitive process a human might undertake when navigating through an unfamiliar landscape with a map. A map reader operates in terms of landmarks, by selecting an easily identifiable point in the landscape and trying to find it in the map, or vice-versa. The cognitive process of comparing two LTSs is similar – easily identifiable states (e.g. states with a distinctive surrounding topology of states and transitions) are used as landmarks, and the rest of the comparison is carried out with respect to them.

The *LTSDiff* algorithm imitates this process. Whereas a human might merely rely on a couple of landmarks for the sake of navigating from one point to another,

Input: LTS_X, LTS_Y, k, t, r
/* LTSs are the two machines, k is the attenuation value, t is the threshold parameter, and r is the ratio of the best match to the second-best score */
Data: *KPairs*, *PairsToScores*, *NPairs*
Result: (*Added*,*Removed*,*Renamed*)
/* two sets of transitions and a relabelling */
1 $PairsToScores \leftarrow computeScores(LTS_X, LTS_Y, k)$;
2 $KPairs \leftarrow identifyLandmarks(PairsToScores, t, r)$;
3 **if** $KPairs = \emptyset$ *and* $S(p_0, q_0) \geq 0$ **then**
4 $\quad KPairs \leftarrow (p_0, q_0)$;
$\quad$ /* p_0 is the initial state in LTS_X, q_0 is the initial state in LTS_Y */
5 **end**
6 $NPairs \leftarrow \bigcup_{(a,b) \in KPairs} Surr(a,b) - KPairs$;
7 **while** $NPairs \neq \emptyset$ **do**
8 $\quad$ **while** $NPairs \neq \emptyset$ **do**
9 $\quad\quad (a,b) \leftarrow pickHighest(NPairs, PairsToScores)$;
10 $\quad\quad KPairs \leftarrow KPairs \cup (a,b)$;
11 $\quad\quad NPairs \leftarrow removeConflicts(NPairs, (a,b))$;
12 $\quad$ **end**
13 $\quad NPairs \leftarrow \bigcup_{(a,b) \in KPairs} Surr(a,b) - KPairs$;
14 **end**
15 $Added \leftarrow \{b_1 \xrightarrow{\sigma} b_2 \in \Delta_Y \mid \nexists\big(a_1 \xrightarrow{\sigma} a_2 \in \Delta_X \wedge (a_1,b_1) \in KPairs \wedge (a_2,b_2) \in KPairs\big)\}$;
16 $Removed \leftarrow \{a_1 \xrightarrow{\sigma} a_2 \in \Delta_X \mid \nexists\big(b_1 \xrightarrow{\sigma} b_2 \in \Delta_Y \wedge (a_1,b_1) \in KPairs \wedge (a_2,b_2) \in KPairs\big)\}$;
17 $Renamed \leftarrow KPairs$;
18 **return** *(Added, Removed,Renamed)*

Algorithm 1: *LTSDiff*

our algorithm wants to make a wholesale comparison between the two machines. For this reason it wants to map *every* feature in one machine to another, starting from landmarks and then exploring the area around them. To do so, it starts off with the most obvious pairs of equivalent nodes, and uses these to compare the surrounding areas, until no further pairs of states or transitions can be found. What is left is the difference between the two machines.

The algorithm is shown in algorithm 1. The functions $computeScores(X, Y, k)$ and $identifyLandmarks(PairsToScores, t)$ were described above. The set of matching state-pairs that surround a given pair (a, b) is identified by the $Surr_{a,b}$ function, which is defined as follows: $Surr_{a,b} = \{(c,d) \in Q_X \times Q_Y \mid \exists \sigma \in (\Sigma_X \cup \Sigma_Y).\ \big((a \xrightarrow{\sigma} c \wedge b \xrightarrow{\sigma} d) \vee (c \xrightarrow{\sigma} a \wedge d \xrightarrow{\sigma} b)\big)\}$.

The algorithm begins with lines 1-5 by finding key-pairs (a one-to-one mapping of states between the two machines) and storing them in $KPairs$ (resorting to using the initial pair of states as the only key-pair if this is unsuccesssful). It then

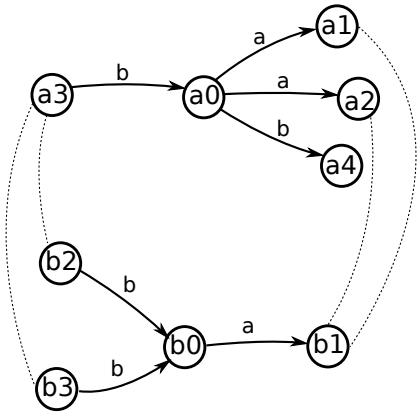


Fig. 4. Two LTSs A and B, with selected pairs of states and their computed scores

proceeds to match up the surrounding transitions for each key-pair, and adds them to a set of candidate matched state pairs $NPairs$ for further exploration (line 6).

The loop in lines 7-14 is responsible for selecting a definitive set of matching pairs from $NPairs$. It begins by iterating through pairs (a, b) from $NPairs$ in the order of their similarity scores (with the *pickHighest* function). Once a pair has been selected, it is added to $KPairs$ (the set of confirmed matches). Due to the potential overlap of multiple incoming / outgoing transitions with the same labels in the subject state-machines, there may be several remaining pairs in $NPairs$ that include either a or b, and these are removed with the *removeConflicts* function. This loop continues until $NPairs$ is empty, when it is recomputed for the enlarged $KPairs$ set (line 13) by considering the surrounding incoming and outgoing transitions for each key pair. Then, the whole process iterates until no further pairs can be added to $NPairs$.

The process of adding pairs into the $NPairs$ set is illustrated with respect to the two machines in Figure 4. Let us assume that $KPairs = \{(a_0, b_0)\}$. Thus, $NPairs = \{(a_3, b_2), (a_3, b_3), (a_1, b_1), (a_2, b_1)\}$. a_4 is not included because there is no outgoing transition from b_0 that is labelled b. Since (a_2, b_1) has the highest score, it is the first to be selected by the $pickHighest$ function and added to $KPairs$. The $removeConflicts$ function removes any of the remaining pairs in $NPairs$ to contain either a_2 or b_1. Consequently, $NPairs = \{(a_3, b_2), (a_3, b_3)\}$. In the following iteration, (a_3, b_3) is added to $KPairs$, and $removeConflicts$ leaves $NPairs$ empty.

4.3.2 *Computing a Patch.* Having computed $KPairs$ in lines 1-14, we use the matching state pairs to identify the delta between the two LTSs, which we refer to as a *patch*.

Definition 4.1 Patch. *For an LTS X, a patch is constructed with two sets of transitions Removed, Added, and one set of state-pairs Renamed that maps states from X to X'. The transitions and states of a patched LTS X' are defined as follows:*
We denote $Q(\Delta)$ as the set of states that are connected by a set of transitions Δ:

$Q(\Delta) = \{q \mid \exists \sigma, p.(p \xrightarrow{\sigma} q \in \Delta) \vee (q \xrightarrow{\sigma} p \in \Delta)\}$. *Thus, the set of transitions in* X' *is computed as follows:* $\Delta_{X'} = Rename(\Delta_X \setminus Removed) \cup Added$ *where Rename denotes an operation replacing all states* q_x *in* $Q(\Delta_X \setminus Removed)$ *with corresponding* q_y *such that* $(q_x, q_y) \in Renamed$. *The set of states* $Q_{X'} = Q(\Delta_{X'})$.

The patch obeys some important properties (proofs for these are available in Appendix C) and is guaranteed to be valid, in that applying the patch to the reference LTS X is guaranteed to result in the subject LTS Y. Secondly, patches are symmetric; applying the inverse of the patch to LTS Y is guaranteed to result in X. Finally, the *LTSDiff* algorithm is robust to the presence of unconnected states in either machine; these do not affect the validity of the patches and are included in them.

4.4 Using the Patch to Measure the Difference between LTSs

LTSDiff returns the structural difference between two LTSs. As with the language comparisons shown in section 3, we would like to be able to attribute some quantitative measurement to this difference. As with the language measurements described in section 3.2, it is possible to construct a confusion matrix. Table VI shows the matrix that can be established once a patch has been computed for a reference LTS R and the subject LTS S.

Impossible (i.e. negative) state transitions are not explicitly represented in LTSs - they are implied by their absence. As such, the set of true negatives TN is always empty. Consequently, we use Precision, Recall and the F-Measure to compare the structural overlap becaues they do not account for true negatives, and will not produce results that are skewed by their absence (in the presence of explicit reject states, alternative measures such as BCR might be preferrable). The formulae have been included in Table II. With respect to LTS structure, Precision will tell us the proportion of transitions in S that are also in R, and Recall will tell us the proportion of transitions in R that are also in S.

4.5 Example: Comparing the Structures of two State Machines of a Text Editor

The remainder of this section illustrates the technique by applying it to the small text-editor example in Figure 1, in a similar vein to the example for language comparison in section 3.3. The structural difference is already visually apparent (the additional `edit` transitions) and this is just for the sake of illustration. The true value of this technique becomes evident for larger-scale machines, where there may be several differences that are not so readily apparent (this will be demonstrated in section 5).

The process begins by constructing the system of linear equations. Using the

	Subject machine S	
Reference machine R	**in** Δ_S	**not in** Δ_S
in Δ_R	$TP = \Delta_R \setminus Removed$	$FN = Removed$
not in Δ_R	$FP = Added$	$TN = \emptyset$

Table VI. Confusion matrix as generated from a patch for two LTSs R and S

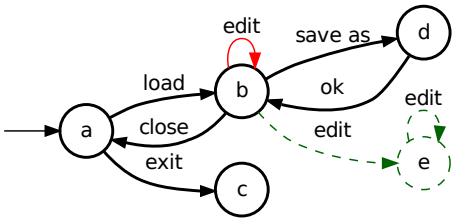


Fig. 5. Diff between correct and incorrect editors - bold edges indicate matching elements, dashed edges indicate added elements, and thin edges indicate missing elements

same process as shown in Figure 3, we apply equation 7. The resulting system of equations given to the solver, as well as the scores given to the states that are deemed to be equivalent are shown in appendix B. As expected, the solver matches the four equivalent states (ae, bf, ch and di), and does not match state g. This means that the resulting transitions attached to G belong to the *Added* set, and the resulting "diff" is shown in Figure 5.

To measure the difference between the two LTSs, it is straightforward to populate the confusion matrix. $|TP| = 5$ (the number of shared edges), $|FN| = 1$ (number of removed edges), and $|FP| = 2$ (number of added edges). Using these measures to compute precision and recall (see Table II), we end up with Precision=0.71, Recall=0.83. This produces an F-Measure of 0.77.

4.6 Implementation and Discussion

The *LTSDiff* algorithm has been implemented as part of our state machine inference and analysis framework StateChum[3]. The user can supply two state machines, and the difference between them is graphically displayed in a window. The linear system of equations is solved with the UMFPACK library [Davis 2004], which takes advantage of the sparse nature of the matrices that are produced. This is combined with the Automatically Tuned Linear Algebra Software (ATLAS) package [Whaley and Petitet 2005], which can be calibrated to take advantage of any specific hardware features such as multicore processors and cache size. With respect to its performance, the most computationally-intensive part of *LTSDiff* is the construction and solution of a system of linear equations. From our empirical evaluation (detailed in Section 6), this does not seem to substantially affect the expense of the overall technique, which seems to scale relatively well.

4.6.1 *Discussion.* *LTSDiff* has a number of advantages over existing comparison techniques (these are discussed in more detail in section 7). Most techniques (c.f. [Sokolsky et al. 2006; Nejati et al. 2007]) require all states to be reachable from the initial state. In addition to this constraint, certain techniques (c.f. Quante and Koschke's approach [Quante and Koschke 2007]) also impose the constraint that the input LTSs must be deterministic and minimal. None of these constraints are necessary for *LTSDiff* , which can just as easily be applied to non-connected, non-minimal and non-deterministic machines.

[3]http://statechum.sourceforge.net

It is possible for *LTSDiff* to produce an inaccurate patch, where most transitions of model X are deemed to be removed and all of model Y are deemed to be added. This may happen if only few key pairs are found or when *identifyLandmarks* chooses too many pairs, and many of them are spurious. In the former case, this would be because the threshold-ratio (t in algorithm 1) is too 'strict' and allows too few pairs to be chosen. On LTSs with disconnected parts, some of these parts may have no state belonging to a key pair, leading the entire part to feature in the *Removed* part and the corresponding part from the second LTS — in the *Added* one. If too many pairs are chosen, many of them will have few 'matched' transitions, with all the remaining ones featuring in *Removed* and *Added*.

The performance of *LTSDiff* can be tuned by adjusting the t, k, and r parameters in algorithm 1. The selection of their values depends to an extent on the characteristics of the state machines. If they are highly homogeneous, with many very similar vertices, the threshold should be higher, to make sure that the algorithm only starts with pairs of states that clearly stand out. Most of the time, states can easily be distinguished from each other. By default (based on experimentation by the authors) the threshold value is set to 50%, ratio of the best match to the second best to 1.4, and the attenuation factor to 0.5.

In practice, the authors have however found that the algorithm is generally accurate, and have not needed to alter the parameters. This will be explored in the empirical study in Section 6.

5. CASE STUDY: THE COMPARATIVE EVALUATION OF TWO REVERSE ENGINEERING TECHNIQUES

There are two perspectives to bear in mind when evaluating the techniques presented in this paper. There is the quantitative aspect, of measuring how scalable the techniques are and analysing the various measures obtained for different types of LTS. There is however also the qualitative aspect; of reasoning about the 'usefulness' of the presented techniques, and discussing their general value in the broader software engineering context. Whereas the next section will be concerned with providing a quantitative analysis, this section provides a case study, demonstrating the value of the two comparison techniques in an LTS reverse-engineering scenario.

As stated in section 1, there are a host of scenarios, especially in the domain of model-driven development, that rely on the comparison of different models. Developers might want to understand how two versions of a system differ from each other, for example Nejati *et al.* [Nejati et al. 2007] motivate their work in the area by comparing two Statecharts of different versions of a call-logging system (one with and one without voicemail). The problem of assessing the accuracy of reverse-engineered models is well established, and has received an increasing amount of attention in recent years [Lo and Khoo 2006; Walkinshaw et al. 2008; Bogdanov and Walkinshaw 2009; Pradel et al. 2010]. Our work is motivated by the need to reliably evaluate and compare different techniques that reverse-engineer state machines, *from different perspectives*. This section will show how the two complementary techniques presented here can be combined to provide a full evaluation of different reverse-engineering techniques.

As a basis for the comparison we use a small model of a CVS client (shown in

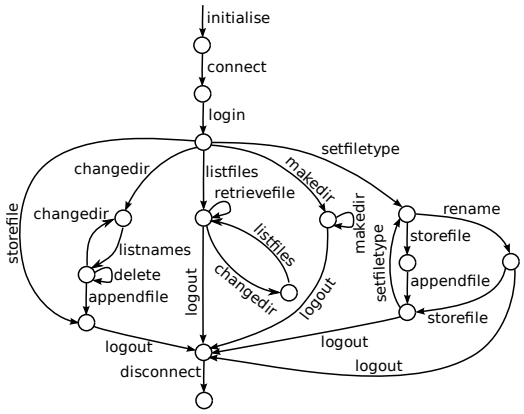


Fig. 6. The original specification of the CVS client

Figure 6 – derived from a similar model presented by Lo *et al.* [Lo and Khoo 2006]). The task will be to use two different reverse-engineering techniques to reverse-engineer this model from a randomised sample of traces (the sample consists of random paths over the model). The different reverse-engineering approaches will produce different models, and the task is to gain insights into the quality of the inferred models, as well as any insights into respective strengths and weaknesses of the two approaches.

The two techniques we use as a basis for this approach we refer to as the *Markov technique* and *Evidence-Driven State Merging (EDSM)*. The Markov technique was proposed by Cook and Wolf [Cook and Wolf 1998] and uses the concept of Markov models to find the most probable state machine from a set of traces. The EDSM approach was developed by Price as a regular grammar inference algorithm that was the winner of the Abbadingo grammar-inference competition [Lang et al. 1998][4].

The models that are reverse-engineered by each technique are presented in Figure 7. From a mere visual comparison of these models to the ideal model in Figure 6 it is very difficult to ascertain which one is more accurate. Therein lies the motivation for our case study. The techniques presented in this paper provide a basis for comparing the models from their behavioural and structural perspectives. This section serves to illustrate how they can be used to gain any insights into the respective strengths and weaknesses of the two reverse-engineering approaches.

5.1 Comparing the Language-Accuracy of the Models

The language-accuracy of the two machines is presented in Table VII. The formulae used to calculate the various measures are presented in Table II. As a reminder, F-Measure is the harmonic mean of Precision and Recall, and BCR is the mean of

[4]The conventional algorithm is designed for two sets of traces; one set of 'valid' sequences, and one set of 'invalid' sequences. Because we are only using a set of traces as input (which we take to represent valid sequences), the algorithm has to be prevented from "overgeneralising" – merging all of the states together to produce a single-state machine. This is achieved by supplying a parameter value, so that it only merges states with a similarity score ≥ 2.

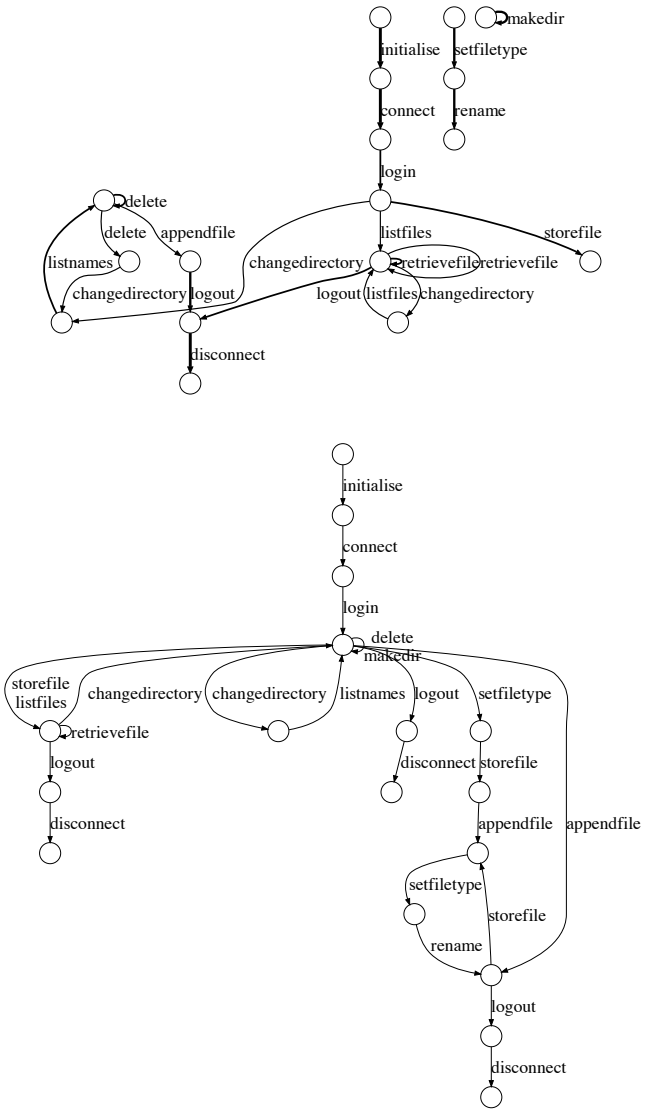


Fig. 7. Inferred models - Markov model at the top and EDSM model at the bottom

Sensitivity (which is the same as Recall) and Specificity.

The scores give us several insights into the behavioural accuracy of the two inferred models. We start with Precision and Recall, which measure the overlap of the languages of the two models. The Markov model has a significantly higher precision but a much lower recall. The difference in Precision leads to a better F-Measure for the Markov approach. These scores indicate that the EDSM model has probably over-generalised from the given sample of the target machine, which

	Precision	Recall (Sensitivity)	F-Measure	Specificity	BCR
Markov	0.948	0.529	0.679	0.997	0.763
EDSM	0.471	0.8	0.593	0.893	0.847

Table VII. Language accuracy results for EDSM and Markov models

has resulted in it producing too many false-positives (but also a greater proportion of true-positives than the Markov model). Based on these scores alone, the Markov model outperforms the EDSM model.

However, if we take the language complement into account as well, things look different. Sensitivity measures the proportion of the language that is contained by the inferred model, Specificity measures the same for the language complement, and BCR is the average of the two. Although Specificity is better for the Markov model, this is compensated by the much better sensitivity score for the EDSM model. Thus, the overall BCR score suggests that, on balance, the EDSM approach is a better result thanks to the fact that it accounts for a greater portion of the target language.

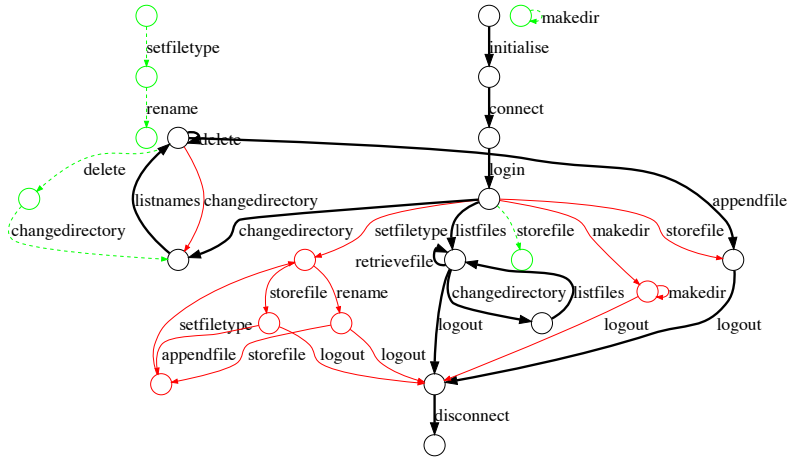
5.2 Comparing the Structural-Accuracy of the Models

By applying the *LTSDiff* algorithm, we can gain more concrete insights comparing the essential components of the inferred machines; their states and transitions. The output from the *LTSDiff* algorithm for both models is presented in Figure 8. Non-bold transitions should be interpreted as "edits" - additions and removals that would be required to transform the inferred model into the reference model. Dashed (light) transitions are additions in the inferred model and non-dashed (dark) transitions are missing. The bold transitions in the *LTSDiff* output are those transitions that are correctly inferred. In this case, 15 out of the 21 edges in the markov model are correct, whereas only 10 of the 21 edges in the EDSM model are correct.

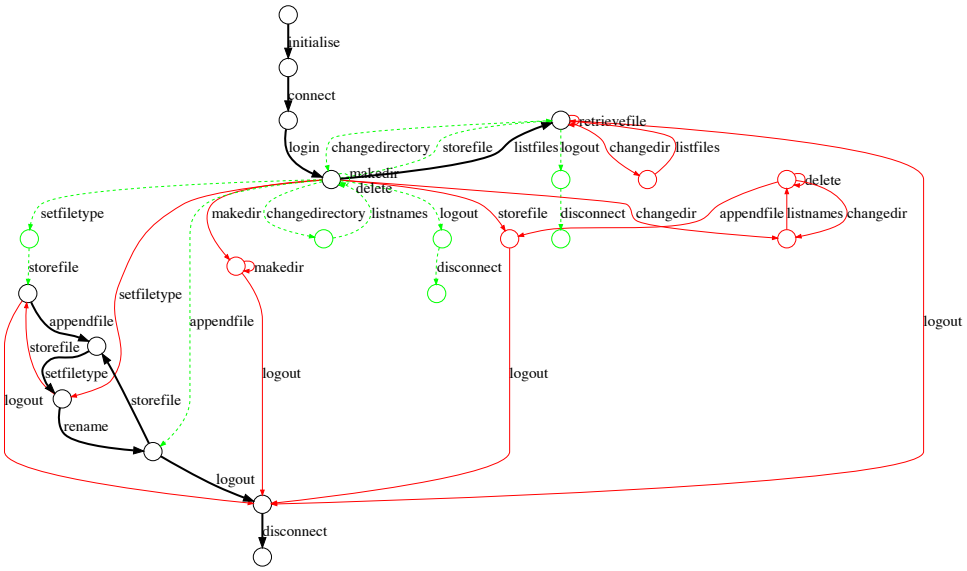
The structural results permit us to see exactly which transitions were (in)correctly inferred by the different techniques. Thus we can establish that there is very little overlap between the two inferred machines; the two different techniques correctly infer different parts of the target model. The patch of correct transitions in the EDSM model starting with the appendfile event is missing from the Markov model, and most of the correct transitions in the Markov model are missing from the EDSM model. This suggests that the two techniques offer complementary strengths, leading to the reasonable conclusion that a hybrid approach would produce better results. This could not have been ascertained from the language-based measures alone.

5.3 The Complementary Nature of the Two Techniques

The ability to compare models in terms of their language and structure is useful because they provide complementary insights into the (dis-)similarity of models. The choice of which one to use depends to a large extent on the context in which the models are to be used. The rest of this subsection briefly discusses two use cases that expose the complementary nature of the two approaches.



Precision: 0.7, Recall: 0.519



Precision: 0.5, Recall: 0.444

Fig. 8. *LTSDiff* results for the Markov result (top) and the EDSM result (bottom), along with the structural precision and recall results (see section 4.4)

For the first use case, imagine that the purpose of the reverse-engineered model is ultimately intended to serve as a test oracle for regression testing. This would enable us to test future versions of the software system against a model representing the current software behaviour. From the testing perspective we are only interested in its externally observable behaviour of the model – its language.

In this case, there is a clear compromise between the two candidate models; for the Markov technique, the language comparison (Table VII) shows us that it is very precise, but only reflects a narrow sample of the behaviour of the software. A test set generated from such a model would therefore similarly be incomplete. The EDSM technique is more complete, but much less precise. A test set generated from this model would wrongly attempt to verify the presence of non-existent behaviour. Thus, judging by the language comparison, it seems intuitive that the Markov model is more useful from a testing perspective, because (though incomplete), the tests that are generated will at least be valid.

For the second use case, we consider the task of qualitatively evaluating reverse-engineered models (this is what actually motivated the work in this paper). In our example scenario, the developer might be disappointed with the performance of the EDSM algorithm, and wish to improve it. Where is the EDSM algorithm going wrong? Why is the EDSM model so inaccurate? Answers to such qualitative questions can be obtained by comparing the structure of the EDSM model to the target to gain an insight into which aspects of the model were inaccurately inferred.

Looking at the *LTSDiff* EDSM output in Figure 8, several problems become apparent. As an example, one basic property of the target model is that all of the `logout` events lead to the same state, which has one final `disconnect` event that terminates the system. The EDSM model adds lots of new individual $\stackrel{logout}{\rightarrow}\stackrel{disconnect}{\rightarrow}$ sequences to the model. This has happened because a limit of 2 was set, to only permit the merging of states with matching suffixes of length 2. As a result, any state that is preceded by a `logout` and followed by a `disconnect` cannot be merged, even though they should. Thus, one way to improve the results would be to adopt a more sophisticated state-merging strategy (one example might be the Blue-Fringe merging strategy [Lang et al. 1998]).

This provides a broad illustration of how the two techniques can be used hand in hand. The language perspective has given us a means to accurately assess the accuracy (in terms of observable behaviour) of two candidate models produced by different reverse-engineering techniques. Subsequently, *LTSDiff* has provided a more qualitative angle, showing us where and why the EDSM-generated model performs poorly, and providing insights as to how this could be improved.

6. EMPIRICAL EVALUATION

The two main factors to affect the performance of both techniques are the complexity of the LTSs in question, and the extent to which they differ. This section presents an empirical study that assesses the performance of the techniques with respect to these factors. Its purpose is to gain an insight into the extent to which the techniques scale, and to compare and contrast the various similarity scores when compared in terms of their structure or language.

6.1 Methodology

In order to manipulate the complexity of the machines in a controlled manner, a random LTS generator [Walkinshaw et al. 2010] was used to produce the machines. These were mutated to varying degrees and compared against the original versions using the two techniques. The comparison processes were monitored to record the amount of time taken, the results for each comparison and, in the case of *LTSDiff* , the extent to which it managed to accurately identify the mutations.

6.1.1 *Random LTSs.* The random LTS algorithm was developed by the authors to form the basis for the STAMINA state machine-inference competition[5], and is implemented as part of the StateChum framework. A detailed discussion of the algorithm is beyond the scope of this paper, but is available in previous work by the authors [Walkinshaw et al. 2010]. In essence, it is designed to generate random LTSs that are similar in character to state-based models of software systems. These tend to have non-uniform edge distributions; certain states are more central to software behaviour, with lots of possible incoming/outgoing transitions, whereas others are merely intermediary states with one incoming and one outgoing transition. Software models tend to have a large alphabet (i.e. the set of possible system events or method calls etc.). Every state is reachable, the transition system is conventionally deterministic, and there is the possibility that a state has transitions that loop back to the same state. The algorithm adapts an existing algorithm by Leskovec *et al.* [Leskovec et al. 2007] that was designed to synthesise complex networks (i.e. with non-uniform edge distributions). The resulting implementation enables us to control the number of states in the random LTSs, along with the size of the alphabet.

6.1.2 *Process.* The random LTS algorithm was used to generate a total of 900 LTSs, which were divided into six sets of 150 LTSs with 20, 70, 120, 170, 220 and 270 states. The size of the alphabet for each LTS was set to half the number of states. It is worth noting that for the sake of this study we are only interested in the general effect of machine complexity on performance, and not the individual effects of alphabet size and machine size, which is why they are tied to each other and not controlled independently.

Each set of 150 LTSs was evenly divided into 5 mutation categories of 30 LTSs, where level 1 represents a slight mutation, and 5 represents a significant mutation. The mutations were carried out as follows. For each LTS a "mutation factor" was calculated by dividing the size of its alphabet by 5. This number multiplied by the mutation category (1 to 5) produces a number that represents the number of mutations to be carried out. A 20 state LTS with an alphabet of 10 would have a mutation factor of 2, which would result in 2 mutations in mutation-category 1, and 10 mutations in category 5, etc. The mutations, picked at random, are listed below:

(1) Add a transition between existing states.

(2) Add a transition to a new state.

[5]http://stamina.chefbe.net/

		Complexity		
		20 states, $\|\Sigma\| = 10$	$\cdots$	270 states, $\|\Sigma\| = 135$
Mutation	1	Simple, lightly mutated	$\cdots$	Complex, lightly mutated
	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	5	Simple, heavily mutated	$\cdots$	Complex, heavily mutated

Table VIII. Tabular representation of random LTSs and their mutations

(3) Remove a transition (will also remove its target state if the state becomes disconnected).
(4) Remove a state.

The variation in model complexity and mutation intensity can be illustrated as a table with six levels of LTS complexity (depending on the number of states), and 5 levels of mutation, as shown in Table VIII.

The comparisons were executed on an Apple Powerbook with a 2.66 GHz Intel Core Duo processor, and 4GB of DDR3 RAM, running Mac OS X 10.6.6.

6.1.3 *Research Questions.* In broad terms, the aim of the study is to establish the scalability and accuracy of the presented techniques. The data should also serve to provide insights into the relationship between the language and structure of state-based model. The specific questions are detailed below.

(1) How expensive are the two techniques?
—To what extent are they affected as models increase in complexity?
(2) How sensitive is the W-Method language comparison technique to mutations?
(3) What is the effect of using the W-Method (or any systematic test set generation technique) to measure language accuracy instead of using a (quasi-)random technique?
(4) How accurate is *LTSDiff* at identifying mutations?
(5) To what extent are the language and structural scores related to each other?

6.1.4 *Random sequence generation.* For the sake of answering question 2, sets of random sequences were generated alongside the sets produced by the W-Method. The random sequence generation algorithm is actually quasi-random. A truly random approach would simply compose sequences as random selections of elements in Σ^*. This would require an impractically large test set before it could be even approximately representative of the subject model [Lang 1992]. Instead, our quasi-random algorithm produces a set of sequences composed of 50% valid and 50% invalid sequences from random walks over the machine. The algorithm makes sure that no sequence is merely a prefix of another sequence, ensuring that every sequence is distinct. Given the non-systematic nature of random sequence generation, it is necessary to specify a limit on the number of sequences. This is computed by multiplying the complexity category (or the column number in Table VIII) by the number of states. So for 20 states it would produce 20 tests, for machines with 170 states (which appear in column 4), it would produce 680 sequences etc.

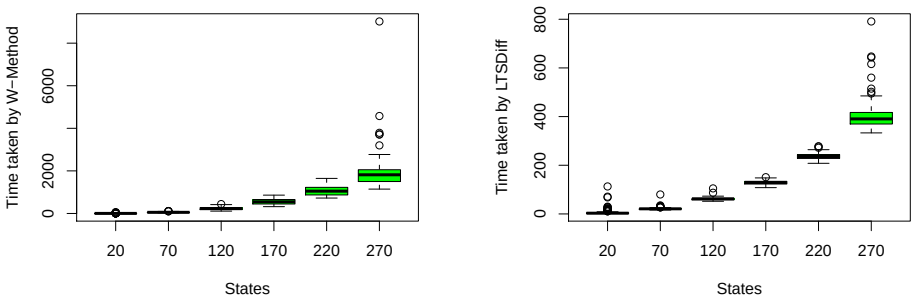


Fig. 9. Average times for LTS comparisons (note the different time-scales on the y-axis)

6.1.5 *Threats to validity.* There are two threats to validity to bear in mind when interpreting these results. These are listed in brief below:

—**Realism of the LTSs:** The LTSs may not accurately reflect the characteristics of realistic software behaviour models. Every effort was taken to minimise this threat by grounding the algorithm to generate them on a broad selection of LTSs for real software systems [Walkinshaw et al. 2010].

—**Equivalent mutations:** There is the danger that individual mutations can undo other mutations, and ultimately lead to an LTS that is very similar to the original. For the smallest model category with 20 states, it is accepted that this might very occasionally happen, and this must be borne in mind when interpreting the results. However, for larger models the probability of this happening is so low that the risk is considered negligible. To account for the possibility that this does occur, the experiment for a given mutation-level / state machine size category is repeated 30 times to ensure that the final findings are not skewed by such possible anomalies.

6.2 Results and Discussion

The full set of charts is available online[6]. For the sake of saving space, this section will only plot and highlight the key results.

6.2.1 *Timing and Scalability.* In terms of timing and scalability, we observe that the language-based comparison (at least when it is based on the W-Method) can become expensive and is much more sensitive to the complexity of the models, whereas the structural comparison scales relatively well and is comparatively cheap. Although particular aspects of performance are discussed by comparing the two techniques, this is merely to provide a point of reference and not to imply that one is "better" than the other – it is important to remember that they are complementary in purpose.

[6] `http://www.cs.le.ac.uk/people/nwalkinshaw/Files/TOSEMResults/performance_charts.zip`

Figure 9 plots the distributions of times to compare the various categories of LTS. The increasingly complex state machines are denoted by their size along the x-axis. It is however important to remember that these machines do not merely vary in terms of the numbers of states, but also in terms of their alphabet size. Each box-plot shows the distribution of the times taken to compute the 150 model comparisons.

It is evident that the language-based comparison is much more expensive, especially for more complex models. The upper quartile for the 270-state machines lies at just below 4000ms for the language-based comparisons, whereas it lies just below 800ms for the *LTSDiff* equivalents. This is explained by the fact that the number of tests produced by the W-Method is exponential with respect to the size and alphabet of the model for which it is produced. Given that the alphabet of the machines with 220 states jumps to 110, the disproportionate increase in the size of the test set means that computing, storing and classifying the test cases suddenly becomes computationally demanding.

6.2.2 *Sensitivity of W-Method language comparison to LTS mutations.* As the level of mutations is increased, it is expected that these will lead to corresponding decreases in the F-Measure that compares the mutated and unmutated LTSs. This relationship is plotted in Figure 10. The plot distinguishes between different mutation levels and different LTS sizes; each box plot is annotated with the mutation levels and the sizes of the models.

The first observation to be made here is that, as expected, the F-Measure faithfully reflects the differences between the models. Low mutation levels produce a high F-Measure, whereas increasing mutation levels produce lower F-Measure scores.

The second observation is concerned with the size of the models. For mutation levels greater than two, the mutations tend to have a higher impact on the F-Measure score for smaller LTSs. This can be explained by the (probable) location of the mutations; mutations that are closer to the initial state of the LTS have a much greater effect than mutations that are further away. For a small LTS, as the number of mutations increases, it becomes increasingly likely that several of these mutations will take place closer to the initial state.

6.2.3 *Comparing W-Method against random samples for computing language-based similarity scores.* To display the extent to which the results tend to vary from each other, the plot in Figure 11 displays the relative distribution of F-Measure results[7] from W-Method and Random algorithms. Each co-ordinate represents the respective scores for a comparison of a single LTS against its mutated counterpart. The accuracy results from the W-Method comparisons are plotted on the x-coordinate, and the corresponding results from the Random sequence set comparisons are plotted on the y-coordinate.

[7]The reader is reminded that in this context the F-Measure is used in a descriptive capacity to show how different two models are, and says nothing about the actual accuracy of the comparison techniques, which is discussed later in this section.

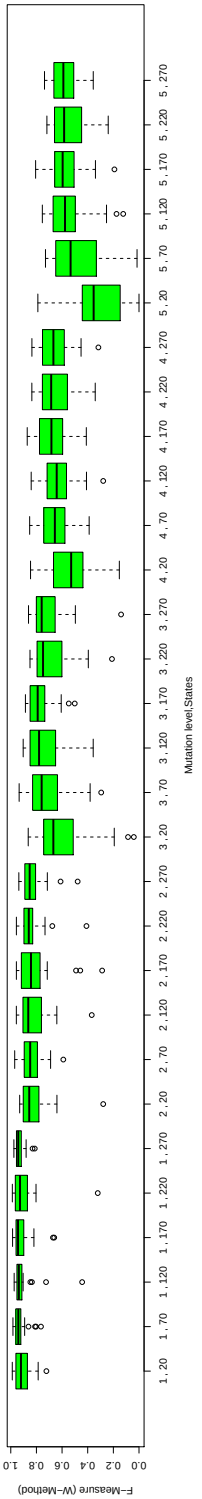


Fig. 10. F-Measure for W-Method with respect to LTS size and mutation factor

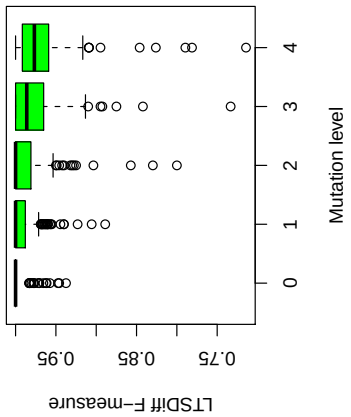


Fig. 12. Accuracy of *LTSDiff* for each mutation level

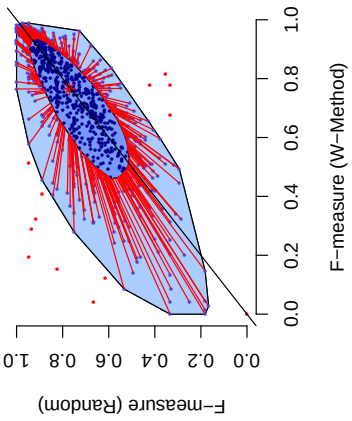


Fig. 11. F-Measure scores for random and W-Method samples

Whereas a conventional scatterplot would hide multiple points with identical coordinates (this is relatively frequent in our case), these are taken into account by the bag-plot [Rousseeuw et al. 1999], a bi-variate extension of the box plot that makes the distribution of all of the coordinates explicit. The inner loop represents the "bag", containing 50% of the points. The "fence" is the outer loop, and represents the upper / lower quartiles in a conventional box plot (i.e. it contains all of the points that are not considered to be outliers). The shape of the bag serves to show the manner in which the results are distributed. A straight line has been superimposed on the plot, to show where the coordinates would have to line up if the results were correlated.

The rationale for the use of the W-Method is that random, non-systematic sampling approaches can produce a misleading sample by failing to incorporate those parts of the machine that are hard to reach. In the plot the bag is asymmetric with respect to the diagonal line, with more of the bag ranging above the line than below it. This indicates that the scores produced with the W-Method tend to be lower, showing that the randomly-generated test sets tend to result in over-approximations of how accurate a model is. This is in large part due to the fact that the Random comparisons tend to produce a higher recall, which is primarily responsible for raising the F-Measure - see Table II. This higher score for the random sequences is misleading; the W-method, with its systematic exploration of the reference model, is generally better at identifying those sequences in the reference model that *should* belong to the language of the subject model but do not. Random walks of a model will mostly contain sequences that are easily identified, and will tend to omit less prominent aspects of the language.

6.2.4 *Accuracy of LTSDiff.* To establish the accuracy of *LTSDiff* with respect to the mutations, we compare mutations with the diff returned by *LTSDiff* . Here we again resort to the F-Measure to compare the outcomes. The box plots in Figure 12 illustrate the distributions of F-Measure scores for each mutation level. These quantify the extent to which the algorithm has correctly identified every added and removed transition.

The size of the LTSs does have a minor bearing on the accuracy of *LTSDiff* . If the algorithm selects a false pair of states as a key pair, the implications for accuracy are greater if the LTS in question is smaller. A substantial proportion of the outliers in Figure 12 stem from comparisons of 20-state LTSs. Accuracy is however slightly affected by the extent to which the model was mutated, which is to be expected, and which is why each mutation level is displayed in a separate box plot.

It is important to bear in mind that this plot only measures the accuracy with respect to a specific "patch". As discussed previously, there can be several different valid patches that would transform one model into another. In this study, it is possible that, even if *LTSDiff* does not identify the exact set of mutations, it still identifies another patch that is equivalent, but expressed in terms of different transitions. For this reason, the accuracy values shown here should be treated as conservative lower-bound estimates.

Even when interpreted in this manner, the results indicate that the *LTSDiff* approach is broadly accurate. For mutation level 1, it is effectively 100% accurate.

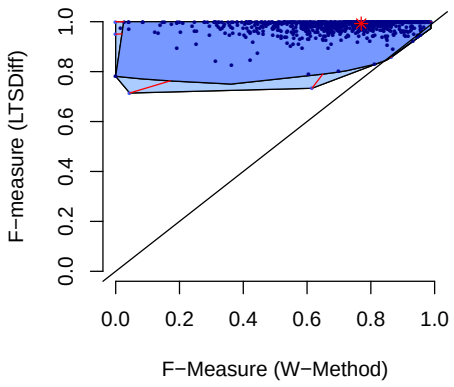


Fig. 13. Bag plot that compares the average F-Measure score for the W-Method against the corresponding score for *LTSDiff*

For each increased level of mutation, the mean accuracy sinks slightly. Ultimately, when heavily mutated (level 5), the lower quartile still lies above 0.95, though some of the lower outliers fall as far as the 0.73 mark.

6.2.5 *Comparing language-accuracy against structure-accuracy.* The two comparison approaches have been presented side-by-side in this paper, because they are intended to provide complementary perspectives on the similarity of different LTSs. This is illustrated when the language and structure similarity scores for a particular pair of models are compared to each other. This is done in Figure 13, where the corresponding structure and language scores for all of the 900 LTS comparisons are plotted as a bag-plot.

The shape of the bag clearly shows that, while the structure of the model maintains its integrity throughout all mutation levels (scores are mostly above 0.8), there is a substantial variance in the language score, which is uniformly distributed from 0 to 1. This is to be expected; a single mutation to the structure of a model (e.g. the removal of an edge) may have a negligible effect on the structure, but can have a disproportionate effect on the language of the model. This highlights the complementary nature of the two perspectives; concentrating solely on the language of a model can provide a misleading view of the extent to which it overlaps with the reference model.

7. RELATED WORK

7.1 Comparing the Languages (Behaviours) of State Machines

In the field of software engineering, most of the work on comparing state machines has focussed on their externally observable behaviour / language. Much of this work has been carried out in the context of model-based testing [Lee and Yannakakis 1996]. The model-based testing problem is based on developing a test set that

will identify a discrepancy between the model and the implementation, which is interpreted as a hidden state machine. This is somewhat different from our problem, where we can see both machines. By identifying a single test that is valid for one machine and invalid for the other, a testing technique has identified a fault. In our case, we want more than that, we already assume that the two machines are different, but we want to quantify this difference and gain insights into the extent to which the languages of the two machines overlap.

Outside of the testing domain, most of the work on comparing state machine languages has been carried out in the field of reverse-engineering. When developing state machine inference techniques, it is important to be able to measure the accuracy of reverse-engineered models in order to be able to evaluate the accuracy of the technique. This is what has motivated the work in this paper and formed the basis for the case study in section 5.

In their work on inferring software process models, Cook and Wolf [Cook and Wolf 1998; 1999] assume that a model has been reverse-engineered from an observed sequence of events. They evaluate their model by observing a further stream of events, and then inducing the closest similar sequence of events in the reverse-engineered model. The task of establishing how accurate the model is, becomes one of measuring the distance between the two event sequences, and they propose a set of metrics to do so. Their setting is different to the one contemplated in this paper; we assume that there are two explicit models, whereas they assume that there is no reference model to compare the inferred model to. In their situation it is hard to evaluate recall, because it is impossible to ascertain the complete set of permissable sequences.

The most common approach for comparing state machine languages explicitly has been relatively straightforward. A random sample of traces is generated which ideally contains an even balance between sequences that should be accepted and sequences that should be rejected. These sequences are then classified by the reference machine, and the proportion of sequences that are correctly classified by the subject machine is taken to be its accuracy. The Abbadingo state machine inference competition [Lang et al. 1998] is a suitable example of this approach, where an inferred model is deemed to be successful if it correctly classifies 99% of the sequences in a random sample. The disadvantages of this approach, the difficulty in randomly producing a suitable set of sequences, and the inability to produce a more descriptive measurement, have been presented in more detail in section 2.

Lo *et al.* [Lo and Khoo 2006] consider a scenario that is more comparable to the situation considered in this paper. They assume that there are two models, one reference and one reverse-engineered, and that these need to be compared. They propose a technique that is based on Precision and Recall. In their setting, two sets of traces are generated; a random sample from the reference LTS, and a random sample from the subject LTS. However, there are several important differences between their work and ours. Firstly, samples are collected differently; they collect *two* samples, one from the correct machine and one from the subject machine. The precision is the proportion of samples generated by the correct machine that are accepted by the subject machine, and the recall is the proportion of sequences that are generated by the subject machine, and are accepted by the correct machine. The

problems associated with random walks have already been discussed. Furthermore, the use of positive traces will only evaluate the machine with respect to its positive language; the absence of invalid traces means that if the machine is too general (falsely accepts too many sequences), this will go unpenalised.

Pradel and Bichsel [Pradel et al. 2010] have also presented a framework for the comparison of state machines in terms of their languages. As with Lo *et al.*, they present the overlap of two models in terms of Precision and Recall. They use an algorithm that was intended to infer models from sequences (the $k-Tails$ algorithm [Biermann and Feldman 1972]). Essentially, the $k-Tails$ algorithm works by constructing a large, highly specific state machine from a set of sequences, and then iteratively generates a smaller, more generalised model by merging states that it deems to be equivalent. Pradel and Bichsel skip the first step, and instead produce an initial model by taking the union of the models they seek to compare. They then use the second $k-Tails$ phase (i.e. the step of identifying similar states) to quantify the similarity of the two models. The use of inference algorithms in this role is interesting. One important difference between the approach presented here and their approach is that, whereas we use the testing algorithms to definitively distinguish between states, algorithms such as $k-Tails$ tend to use heuristics that can be highly inaccurate [Walkinshaw et al. 2008]. On the other hand, algorithms such as W-Method tend to suffer a large performance impact on larger, more complex models, whereas one can envisage that algorithms such as Pradel and Bichsel's would scale more easily.

7.2 Structural State Machine Comparison

7.2.1 *Quantitative bisimulation.* Sokolsky *et al.* [Sokolsky et al. 2006] proposed a graph-comparison technique that is based upon the notion of 'quantitative bisimulation'. This has formed the basis for the comparison of UML Statecharts by Nejati *et al.* [Nejati et al. 2007]. They assume that the reference machine obeys the constraint that all states can be reached from the initial node, and that it is deterministic (for every label in one machine, the algorithm will only attempt a single corresponding transition in the other machine). Their algorithm starts the machine comparison at the initial states of the two machines. Scores are propagated through the machine by choosing best next pairs until the process converges at a fixed point.

The notion of quantitative bisimilarity is interesting, because it approaches the problem of state-similarity from a different perspective to the landmark-approach proposed in this paper. Although the graph-comparison approach itself suffers from an important vulnerability that is addressed by our approach, there remains the possibility of integrating the notion of quantitative bisiumlation into the *LTSDiff* approach. These are discussed briefly below.

The vulnerability is rooted in the fact that the similarity scores for a given pair of states can only be valid if the pairs of states that were identified in the run-up are themselves equivalent. In the scenario where disparate areas of the machine that may be far removed from the initial node are similar, but the interim states and transitions are completely different, this approach can easily produce unreliable results. This problem is acknowledged by Sokolsky *et al.* in their conclusions, who state the aim of identifying areas of high similarity within graphs, instead of

attempting to compare graphs wholesale.

The *LTSDiff* technique presented here offers some important improvements. *LTSDiff* ensures that the computation of a score for a pair of states is not dependent on a specific path used to reach the pair from the initial state. Thus, even in cases where only small isolated areas of the machine are very similar to their counterparts in the other machine, these are more reliably identified by our approach.

There does however remain the possibility of combining the two approaches. Although not as suited for comparing models on a global basis, the measure does provide a potentially useful basis for comparing the similarity of individual states, as required for the identification of key pairs by *LTSDiff* . The current approach used by *LTSDiff* is capable of mismatching key pairs. As discussed in section 4.6, one of the tasks in our future work will be to look at alternative measures for comparing the localised behaviour of individual states – in this respect quantitative bisimilarity could be a useful measure that could, depending on the model characteristics, provide more accurate matches.

7.2.2 *Other approaches.* Melnik *et al.* [Melnik et al. 2002] also adopt fixed-point computation to match states. In a similar fashion to the attenuation factor in our algorithm, they adopt the notion of state-dependent 'propagation coefficient'. They adopt a useful filtering approach that incorporates several different techniques in order to approximate results that would be obtained if a human was comparing two state machines. The integration of a range of these filtering techniques of [Melnik et al. 2002] will be investigated in our future work.

In their work on evaluating reverse-engineered automata, Quante and Koschke [Quante and Koschke 2007] present a technique with a similar aim to ours. Their approach works by computing the union of the two machines [Hopcroft et al. 2007], and comparing each machine to the union (by computing the product of each machine with the union). Computing the product produces a count of missing and inserted transitions. In practice, there are a number of problems that can arise with this technique. Computation of a union produces a non-minimal state machine that is non-deterministic. Removing this nondeterminism and minimising the non-deterministic machine can result in a very large machine even if there are only few differences between the subject and reference machines. In other words, there is the danger that the computed "patch" (added and removed transitions) could be much bigger than it needs to be.

The problem of comparing state machines shares several traits with the protein sequence-alignment problem [Needleman and Wunsch 1970]. The problem of finding an alignment of proteins that maximises their overlapping segments is similar to the challenge of finding an overlap of state machines that maximises the overlap of shared transitions. Solutions such as the Needleman Wunsch algorithm address this challenge in a similar way to the *LTSDiff* algorithm presented in this work, by identifying 'landmarks' to detect overlapping states / protein segments.

The problem of computation of maximal set of common edges between two undirected graphs has been considered by [Raymond et al. 2002]; their approach has some similarity to this work. First of all, both focus on identification of a maximal set of 'unchanged' edges. The 'second-tier' approximation of the solution size in

[Raymond et al. 2002] uses a matrix of scores for pairs of states, similar to our local similary scores. From this matrix, best matches are picked; the main algorithm of [Raymond et al. 2002] is a search with backtracking. In our work, pairwise scores are computed recursively and many matches can be filtered out in an attempt to find best matches. The fact that *LTSDiff* considers directed graphs means that (1) the outcome of a recursive score computation is of good quality and (2) *LTSDiff* can use these scores and avoid backtracking.

The challenge is a specific instance of the graph isomorphism problem, called error-correcting graph isomorphism [Bunke 1997]. Conventional approaches to computing the edit-distance between graphs take as input two undirected graphs. In our case, node comparison relies on edge labels and edge direction.

In the context of inference of probabilistic automata, work by Kermorvant and Dupont [Kermorvant and Dupont 2002] describes an inference process, using a procedure similar to the idea of local similarity. Given (1) probabilities of transitions from each state, (2) probabilities that a state is reached and (3) probabilities that a sequence reaching each state terminates there, the procedure described in [Kermorvant and Dupont 2002] determines when two states can be considered probabilistically behaviourally-equivalent.

8. CONCLUSIONS AND FUTURE WORK

We have presented two techniques that enable the comparison of LTSs from two complementary perspectives. To compare LTSs from a language-perspective, we leverage the ability of model-based testing techniques such as the W-Method to obtain a finite set of sequences that represent the complete spectrum of behaviour for a given LTS. Such a set forms an authoritative basis for establishing the language-similarity of two LTSs. In terms of their structure, we have presented a technique that identifies a subset of states that are deemed to be equivalent, and then uses these as the basis for working out the difference between two LTSs in terms of their states and transitions. In both cases, the differences are measured in terms of Precision and Recall [Rijsbergen 1979]. This measure establishes the extent to which the two languages / transition structures overlap, and provides an insight into how two LTSs differ. All of the approaches discussed in this paper have been implemented, and the source code can be downloaded as part of the StateChum tool [8].

The work has been primarily motivated by the authors' work on reverse-engineering state-machines from program traces [Walkinshaw et al. 2007]. As shown in the case study, the ability to compare inferred state machines to each other, or to some reference model, is crucial if they are to be evaluated properly. The two techniques described in this paper form a useful basis for such an evaluation.

With respect to the work on measuring the differences in terms of LTS language, our future work will investigate the use of alternative test set generation techniques. We aim to investigate alternative conformance testing techniques [Fujiwara et al. 1991; Lee and Yannakakis 1996]. Although the use of systematic testing techniques, coupled with a more detailed measurement approach, is undoubtedly more reliable than random testing, one important aspect of our future work will be to establish

[8] http://statechum.sourceforge.net

the extent to which the use of alternative test generation techniques changes the similarity measurements.

Although the two techniques are intended to work side-by-side, our future work will investigate the potential for a more integrated approach. One potentially problematic factor with the structural comparison approach is the fact that, for large LTSs, the computation of initial state-pair scores can become too computationally expensive. One way to address this problem would be to use the language-similarity of each pair of states instead.

Acknowledgements

The authors thank the anonymous reviewers for their valuable comments and suggestions. They also thank Jonathan Cook at the University of New Mexico for providing his implementation of the Markov state machine inference tool, which formed the basis for our case study. The idea that Specificity and BCR would form a suitable basis to measure the accuracy of LTSs with respect to their language complement stems from discussions with Pierre Dupont at the Universite Catholique de Louvain. Finally, the authors thank the anonymous reviewers for their valuable and constructive feedback.

REFERENCES

Biermann, A. and Feldman, J. 1972. On the Synthesis of Finite-State Machines from Samples of their Behavior. *IEEE Transactions on Computers 21*, 592–597.

Bogdanov, K., Holcombe, M., Ipate, F., Seed, L., and Vanak, S. 2006. Testing methods for X-Machines: A review. *Formal Aspects of Computer Science 18*, 3–30.

Bogdanov, K. and Walkinshaw, N. 2009. Computing the Structural Difference between State-Based Models. In *16th IEEE Working Conference on Reverse Engineering (WCRE)*.

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

Bunke, H. 1997. On A relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters 18,* 8 (Aug.), 689–694.

Cheng, K. and Krishnakumar, A. 1993. Automatic functional test generation using the extended finite state machine model. In *30th ACM/IEEE Design Automation Conference*. 86–91.

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

Clarke, E., Grumberg, O., and Peled, D. 1999. *Model Checking*. The MIT Press, Cambridge, Massachusetts.

Cook, J. and Wolf, A. 1998. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology 7,* 3, 215–249.

Cook, J. and Wolf, A. 1999. Software process validation: Quantitatively measuring the correspondence of a process to a model. *ACM Transactions on Software Engineering and Methodology 8,* 2, 147–176.

Davis, T. 2004. Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw. 30,* 2, 196–199.

Fujiwara, S., v. Bochmann, G., Khendek, F., Amalou, M., and Ghedamsi, A. 1991. Test selection based on finite state models. *IEEE Transactions on Software Engineering 17,* 6 (June), 591–603.

Gill, A. 1962. *Introduction to the Theory of Finite State Machines*. McGraw-Hill.

Harel, D. and Naamad, A. 1996. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology 5,* 4, 293–333.

Hopcroft, J., Motwani, R., and Ullman, J. 2007. *Introduction to Automata Theory, Languages, and Computation, Third Edition*. Addison-Wesley.

Kelter, U. and Schmidt, M. 2008. Comparing state machines. In *Comparison and Versioning of Software Models (CVSM 2008)*.

Kermorvant, C. and Dupont, P. 2002. Stochastic grammatical inference with multinomial tests. In *Grammatical Inference: Algorithms and Applications*. Lecture Notes in Artificial Intelligence, vol. 2484. Springer-Verlag, 149–160.

Lang, K. 1992. Random DFA's can be approximately learned from sparse uniform examples. In *Proceedings of the International Conference on Learning Theory (COLT'92)*. 45–52.

Lang, K., Pearlmutter, B., and Price, R. 1998. Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm. In *Proceedings of the International Colloquium on Grammar Inference (ICGI)*. Vol. 1433. 1–12.

Laycock, G. 1992. Formal Specification and Testing: A Case Study. *Software Testing, Verification and Reliability 2(1)*, 7–23.

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

Leskovec, J., Kleinberg, J., and Faloutsos, C. 2007. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data 1,* 1.

Lo, D. and Khoo, S. 2006. QUARK: Empirical assessment of automaton-based specification miners. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'06)*. IEEE Computer Society, 51–60.

Melnik, S., Garcia-Molina, H., and Rahm, E. 2002. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In 18th International Conference on Data Engineering (ICDE 2002). *Proc. 18th ICDE Conf. (Best Student Paper award)*.

Needleman, S. and Wunsch, C. 1970. A general method applicable to the search of similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology 48*, 443–453.

Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., and Zave, P. 2007. Matching and merging of statecharts specifications. In *International Conference on Software Engineering (ICSE'07)*. IEEE Computer Society, 54–64.

Pradel, M., Bichsel, P., and Gross, T. 2010. A framework for the evaluation of specification miners based on finite state machines. In *Proceedings of the International Conference on Software Maintenance (ICSM'10)*. Timisoara, Romania.

Quante, J. and Koschke, R. 2007. Dynamic protocol recovery. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE'07)*. 219–228.

Raymond, J., Gardiner, E., and Willett, P. 2002. RASCAL: Calculation of graph similarity using maximum common edge subgraphs. *The Computer Journal 45,* 6, 631–644.

Rijsbergen, C. J. V. 1979. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA.

Rousseeuw, P., Ruts, I., and Tukey, J. 1999. The bagplot: A bivariate boxplot. *The American Statistician 53,* 4 (Nov.), 382–387.

Sokolova, M. and Lapalme, G. 2009. A systematic analysis of performance measures for classification tasks. *Inf. Process. Manage 45,* 4, 427–437.

Sokolsky, O., Kannan, S., and Lee, I. 2006. Simulation-based graph similarity. In *In Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006*. Lecture Notes in Computer Science, vol. 3920. Springer, 426–440.

Sorrows, M. and Hirtle, S. 1999. The nature of landmarks for real and electronic spaces. In *Spatial information theory - Cognitive and computational foundations of geographic information science*. Springer, Berlin, 37–50.

Vasilevskii, M. 1973. Failure diagnosis of automata. *Cybernetics and Systems Analysis*.

Walkinshaw, N., Bogdanov, K., Damas, C., Lambeau, B., and Dupont, P. 2010. A framework for the competitive evaluation of model inference techniques. In *Proceedings of the International Workshop on Model Inference In Testing (MIIT'10)*.

Walkinshaw, N., Bogdanov, K., Holcombe, M., and Salahuddin, S. 2007. Reverse Engineering State Machines by Interactive Grammar Inference. In *14th IEEE International Working Conference on Reverse Engineering (WCRE)*.

Walkinshaw, N., Bogdanov, K., Holcombe, M., and Salahuddin, S. 2008. Improving Dynamic Software Analysis by Applying Grammar Inference Principles. *Journal of Software Maintenance and Evolution: Research and Practice*.

Walkinshaw, N., Bogdanov, K., and Johnson, K. 2008. Evaluation and Comparison of Inferred Regular Grammars. In *Proceedings of the International Colloquium on Grammar Inference (ICGI)*. St. Malo, France.

Walkinshaw, N., Derrick, J., and Guo, Q. 2009. Iterative refinement of reverse-engineered models by model-based testing. In *Proceedings of Formal Methods (FM'09)*. LNCS, vol. 5850. Springer, 305–320.

Weyuker, E. 1983. Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems 5,* 4, 641–655.

Whaley, R. and Petitet, A. 2005. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience 35,* 2 (February), 101–121.

A. TEST SET FOR TEXT EDITOR

In the following table, the first two columns represent whether the test is accepted or rejected by subject machine S or reference machine R (0 for reject and 1 for accept).

Subject Machine S	Reference Machine R	Test
0	0	exit, saveas
0	0	exit, edit
0	0	exit, load
0	0	exit, ok
0	0	exit, exit
0	0	exit, close
0	0	load, saveas, saveas
0	0	load, saveas, edit
0	0	load, saveas, load
1	1	load, saveas, ok, saveas, ok
0	0	load, saveas, ok, saveas, exit
0	0	load, saveas, ok, saveas, close
0	0	load, saveas, ok, edit, ok
0	0	load, saveas, ok, edit, exit
0	1	load, saveas, ok, edit, close
0	0	load, saveas, ok, load
0	0	load, saveas, ok, ok
0	0	load, saveas, ok, exit
0	0	load, saveas, ok, close, ok
1	1	load, saveas, ok, close, exit
0	0	load, saveas, ok, close, close
0	0	load, saveas, exit
0	0	load, saveas, close
0	1	load, edit, saveas, ok
0	0	load, edit, saveas, exit
0	0	load, edit, saveas, close
0	0	load, edit, edit, ok
0	0	load, edit, edit, exit
0	1	load, edit, edit, close
0	0	load, edit, load
0	0	load, edit, ok
0	0	load, edit, exit
0	0	load, edit, close, ok
0	1	load, edit, close, exit
0	0	load, edit, close, close
0	0	load, load
0	0	load, ok
0	0	load, exit
0	0	load, close, saveas
0	0	load, close, edit
0	0	load, close, load, ok
0	0	load, close, load, exit
1	1	load, close, load, close
0	0	load, close, ok
0	0	load, close, exit, ok
0	0	load, close, exit, close
0	0	load, close, close
0	0	close
0	0	ok
0	0	edit
0	0	saveas

B. SYSTEM OF EQUATIONS FOR TEXT EDITOR

	ea	eb	ec	ed	fa	fb	fc	fd	ga	gb	gc	gd	ha	hb	hc	hd	ia	ib	ic	id	=
ea	4					-k									-k						2
eb		10																			0
ec			4																		0
ed				6																	0
fa					10																0
fb	-k					6				-k										-k	3
fc							6														0
fd								8													0
ga									6												0
gb										6-k											1
gc											2										0
gd												4									0
ha													4								0
hb														6							0
hc															1						0
hd																2					0
ia																	6				0
ib																		8			0
ic																			2		0
id						-k														2	1

Scores for matched pairs: AE=12, BF=11, CH=6, DI=13.

C. PROOFS FOR PATCH SYMMETRY AND VALIDTY

THEOREM C.1 SYMMETRY OF THE PATCH. *Given LTS X and Y,* LTSDiff (X, Y) *can return* (*Added*, *Removed*, *Renamed*) *if and only if* LTSDiff (Y, X) *can return* (*Removed*, *Added*, *Renamed*$^{-1}$). *Note that since Renamed is a one-to-one function, so will be Renamed*$^{-1}$.

PROOF. We need to show that *LTSDiff* (Y, X) can compute *Renamed*$^{-1}$ (for instance, due to non-determinism in the subject machines, *LTSDiff* may compute different outcomes). The symmetry follows from the definitions of *Removed* and *Added*.

In order to demonstrate that *Renamed* from *LTSDiff* (X, Y) is the same as *Renamed*$^{-1}$ from *LTSDiff* (Y, X), we have to prove that construction of *KPairs* is symmetrical. This can follows from the following points:

(1) Computation of scores $S(X, Y)$ is symmetric. Hence, $computeScores(x, y, t, r) = computeScores(x, y, t, r)^{-1}$.
(2) The selection of pairs from a collection of them by both *identifyLandmarks* and *PickHighest* only considers scores rather than properties of specific states. Among the possible outcomes of *PickHighest* there is a symmetric one. For this reason, line 6 of algorithm 1 can deliver a symmetric *KPairs* set.
(3) Lines 6 and 13 compute *NPairs* symmetrically.
(4) The *removeConflicts* operation is symmetric.

The above shows that at line 15 *KPairs* of *LTSDiff* (X, Y) is the same as $KPairs^{-1}$ of *LTSDiff* (Y, X). □

THEOREM C.2 PATCHES ARE ALWAYS VALID. *Assume that:*

(1) *Names of states in LTSs X and Y do not intersect (this can be accomplished using a suitable renaming) and consider an LTS BIG consisting of all states in* Q_X *and* Q_Y.

(2) *Before a patch is applied,* $\Delta_{BIG} = \Delta_X$.

(3) *Renamed* $: Q_X \to Q_Y$ *is a one-to-one function.*

(4) *If (1)-(3) are satisfied, the application of the patch computed by the* LTSDiff *algorithm yields* $\Delta_{BIG} = \Delta_Y$ *regardless of the choice of pairs in Renamed.*

PROOF. If there are no key pairs found, $Added = \Delta_X$, $Removed = \Delta_Y$ and $Renamed = \emptyset$, hence $\Delta_{BIG} = (\Delta_X - \Delta_X) \cup \Delta_Y = \Delta_Y$.

If a number of key pairs are found, *Renamed* contains these key pairs. By definition of *Removed*, transitions *not* included in *Removed* are transitions between key pairs described as: $N = \{a_1 \xrightarrow{\sigma} a_2 \mid \exists a_1, a_2 \in Q_X; b_1, b_2 \in Q_Y.\big((a_1, b_1) \in KPairs \wedge (a_2, b_2) \in KPairs \wedge a_1 \xrightarrow{\sigma} a_2 \in \Delta_X \wedge b_1 \xrightarrow{\sigma} b_2 \in \Delta_Y\big)\}$. These transitions will remain after those in *Removed* are taken out. The renaming process turns all q_X to q_Y where $(q_X, q_Y) \in Renamed$; moreover, all states in N are mapped to some states in Q_Y by *Renamed*. Since $Renamed = KPairs$, after making the renaming substitutions in the above expression, we get a set of transitions $\{b_1 \xrightarrow{\sigma} b_2 \mid \exists a_1, a_2 \in Q_X, b_1, b_2 \in Q_Y.\big((a_1, b_1) \in KPairs \wedge (a_2, b_2) \in KPairs \wedge a_1 \xrightarrow{\sigma} a_2 \in \Delta_X \wedge b_1 \xrightarrow{\sigma} b_2 \in \Delta_Y\big)\}$. By comparing this to the definition of *Added*, this is exactly the set of transitions from Δ_Y which is *not* going to be added. This shows that after applying a computed 'patch' the outcome contains exactly the transitions of LTS B. □

It is straightforward to extend the algorithm to incorporate states which are completely unconnected. For an LTS X, these can be defined as $UC_X = \{q \in Q_X \mid \nexists p \in Q_X, \sigma \in \Sigma_X.(q \xrightarrow{\sigma} p \in \Delta_X) \vee (p \xrightarrow{\sigma} q \in \Delta_X)\}$. If both LTSs under comparison contain such vertices, pairs of them can be added to *Renamed*. If this is not possible (e.g. one LTS has more unconnected states than the other) the unmatched unconnected states can be handled by adding them to a set $AddedUV_{XY}$, which can be incorporated in to the patch, where $AddedUV_{XY} = \{b \in UC_Y \mid \nexists a \in Q_X.(a, b) \in Renamed\}$. A revised patch can hence be defined as the quadruple $(Added, AddedUV, Removed, Renamed)$. The patch is applied as above, except that the states in the patched LTS X' are defined as: $Q_{X'} = \{q \mid \exists p, \sigma.p \xrightarrow{\sigma} q \in \Delta_{X'} \vee q \xrightarrow{\sigma} p \in \Delta_{X'}\} \cup \{q \mid \exists p \in Q_X.(p, q) \in Renamed\} \cup AddedUV$. In other words, the states in $Q_{X'}$ consist of those mapped by *Renamed* to states of X contained in $\Delta_{X'}$, *Renamed*, or *AddedUV*.

THEOREM C.3 CORRECTNESS OF PATCH WITH UNCONNECTED STATES. *Given LTSs X and Y, an extended patch* $(Added, AddedUV, Removed, Renamed)$ *as described above will produce Y when applied to X.*

PROOF. The computation of Δ is unaffected by unconnected states, hence the result of theorem C.2 still holds. By construction of *Renamed*, for any pair $(p, q) \in Renamed$, $q \in Q_Y$. The definition of $Q_{X'}$ can be rewritten as follows:

$$Q_{X'} = Q(\Delta_Y) \cup \{q \in Q_Y \mid \exists p \in Q_X.(p,q) \in Renamed\} \cup AddedUV_{XY}$$

Can be rewritten as (a) $\cup$ (b), where:

$$\overbrace{\{q \in UC_Y \mid \exists p \in Q_X.(p,q) \in Renamed\}}^{(a)}$$

$$\overbrace{\{q \in Q_Y \setminus UC_Y \mid \exists p \in Q_X.(p,q) \in Renamed\}}^{(b)}$$

Can be rewritten as (c):

$$\overbrace{\{b \in UC_Y \mid \nexists a \in Q_X.(a,b) \in Renamed\}}^{(c)}$$

Since $Q(\Delta_Y) = Q_Y \setminus UC_Y$, by definition of UC_Y, $(b) \subseteq Q(\Delta_Y)$, additionally $(a) \cup (c) = UC_Y$. Hence, $Q_{X'} = Q(\Delta_Y) \cup (a) \cup (b) \cup (c) = Q(\Delta_Y) \cup UC_Y = Q_Y$.

□