Computing the Structural Difference between State-Based Models

Kirill Bogdanov and Neil Walkinshaw
Department of Computer Science
The University of Sheffield
Sheffield, UK
{*k.bogdanov,n.walkinshaw*}*@dcs.shef.ac.uk*

***Abstract*—Software behaviour models play an important role in software development. They can be manually generated to specify the intended behaviour of a system, or they can be reverse-engineered to capture the actual behaviour of the system. Models may differ when they correspond to different versions of the system, or they may contain faults or inaccuracies. In these circumstances, it is important to be able to concisely capture the differences between models – a task that becomes increasingly challenging with complex models. This paper presents the PLTSDiff algorithm that addresses this problem. Given two state machines, the algorithm can identify which states and transitions are different. This can be used to generate a 'patch' with differences or to evaluate the extent of the differences between the machines. The paper also shows how the Precision and Recall measure can be adapted to quantify the similarity of two state machines.**

I. Introduction

Models of software behaviour are useful for a variety of development tasks. Developers can use them as a basis for communicating with each other, they can be inspected, or used for specification-based testing or model checking. If the model is complete and up-to-date, the aforementioned techniques can be used in harmony to ensure that the software ultimately behaves correctly.

The need to authoritatively compare two different models arises in a number of areas in software engineering. For example, the accuracy of reverse-engineering techniques can be evaluated by examining differences between models they produce from the same system. In model-driven development, a developer will inevitably end up with multiple models corresponding to different versions, alternative specifications, or reverse-engineered implementations. Being able to concisely characterise the differences between these models is key to developing the system in such a way that it is both correct and consistent.

Traditionally, model-comparison techniques have been black-box approaches that compare the observable "languages" of two models, where the language is the set of all possible (and impossible) sequences of events in a model (c.f. previous work by the authors [1], [2]). Although this sort of comparison presents useful insights into the external behaviour of the model, it neglects its actual transition structure. As a motivating example, a developer may look at two models corresponding to different versions of a software system, and try to understand what has changed. Comparing them in terms of their languages will not fully answer this question. What is required is a white-box approach that manages to consisely capture which states and transitions have been inserted or removed to transform one model into the other.

This paper makes two contributions. It presents the PLTSDiff algorithm, which concisely captures the difference between two models in terms of added / removed states and transitions. It also shows how the output of the algorithm can be used with the well established precision and recall measure [3] to provide a more descriptive measure of similarity between two models. The algorithm supports non-deterministic models containing disconnected parts.

The rest of this paper is structured as follows. Section II introduces the challenge of comparing finite state models, and shows why existing techniques are insufficient. Section III introduces the core of our technique - a mechanism that computes the similarity of states in different machines, based on their surrounding transition structure. Section IV presents our comparison algorithm that is based on the state-similarity scores, and also shows how the output of the algorithm can be used to compute the precision and recall measure. Section V presents a case study that demonstrates the application of the algorithm by comparing the outputs from two different state machine inference techniques. Finally, sections VI and VII discuss related work and outline our future plans, respectively.

II. Background

This section presents some preliminary definitions, and then provides an overview of existing techniques that are used to compare state-based models. These techniques are primarily from the domain of finite state machine / regular grammar inference, where they are used to establish the accuracy of inferred models with respect to reference models.

A. Preliminary definitions

In this work, we assume that software behaviour is modelled in terms of a *Labelled Transition System*. A broad range of state-based formalisms can be interpreted as LTSs (e.g. Abstract State Machines [4], Extended Finite-State Machines [5]). Ultimately, an LTS represents the set of all

valid and invalid sequences of events or functions (which are represented by labels).

Definition 2.1 (Labelled Transition System (LTS)): A LTS [6] is a quadruple (Q, Σ, Δ, q_0), where Q is a finite set of states, Σ is a finite alphabet, $\Delta : Q \times \Sigma \rightarrow Q$ is a partial function and $q_0 \in Q$. 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.

State-based descriptions of software behaviour are usually presumed to be complete; if a particular transition cannot be followed in a model, it is assumed to be impossible. In practice however, models are usually incomplete. Whether designed by hand or reverse-engineered, it is rarely realistic to assume that a model is complete in every respect. As a result, the absence of a transition may be either due to the fact that it would be impossible, or that it is simply not known. To accomodate this possibility, the technique presented in this paper is defined with respect to an extension of conventional LTS's. We interpret the models to be compared as Partial LTSs (PLTSs) [6] which allow one to explicitly distinguish between model transitions that are known to be invalid, and transitions that are simply not known to exist at all.

Definition 2.2 (Partial LTS (PLTS)): A PLTS is a 5-tuple $(Q, \Sigma, \Delta, q_0, \Psi)$. This is defined as a LTS, but it is assumed to be only partial. Let the set of elements of an alphabet used on outgoing transitions from a specific state s can be denoted by $\Sigma_s = \{\sigma \in \Sigma | \exists t \in Q . s \xrightarrow{\sigma} t \in \Delta\}$. To make the distinction between unknown and invalid behaviour, function $\Psi : Q \rightarrow 2^{\Sigma}$ is introduced; it has to satisfy $\forall q \in Q . \Sigma_q \cap \Psi_q = \emptyset$.

Definition 2.3 (Language of a PLTS): The sequence $\alpha \in \Sigma^*$ belongs to the *language* of a PLTS if there is a path from the initial state q_0 to a state q. In symbols, $q_0 \xrightarrow{\alpha} q$, for $q \in Q$.

B. The Challenge of Structural Comparison of Finite-State Models

When comparing or evaluating models of software behaviour, it is important to account for the structure of the model, as well as its language. Ideally, there should be a measure of *structural* overlap between two models, that can be presented alongside the conventional language-based similarity measures. One intuitive approach is to look at the difference between two models in terms of the set of states and transitions that are superfluous or missing. This is reminiscent of Levenshtein's edit-distance metric [7], but instead of strings, we are comparing state machines.

Structurally comparing two non-trivial LTS's 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. An LTS is a simplification of the richer Statechart-like notations that are often used in software engineering. Data-related annotations are stripped away; states are mere points in time that are used to specify a partial order of events as specified by the state transition labels. Accordingly, we cannot rely on state-annotations to compare models. Any two transitions in the two machines that have the same label 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.

III. Measuring the Similarity of States

The PLTSDiff algorithm (which is described in section IV) depends on the ability to compute a score that measures the similarity 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 III-A describes how two states can be compared in terms of the overlap of their adjacent transitions, section III-B then gives a recursive extension accounting for similarity of their adjacent states.

A. Scoring local similarity

In this subsection we describe the process of calculating local similarity of two states for non-deterministic PLTS. 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, $S_{AB} = \frac{|(matchingAdjacentTransitions)|}{|(AllAdjacentTransitions)|}$.

The similarity of two states can be computed either in terms of their outgoing or incoming transitions. We start by describing how to match states in terms of their outgoing transitions; the analogous computation for incoming transitions is very similar. Section III-B describes how to compute the global similarity of all pairs of states will also show how the two outgoing and incoming measures are combined to form a single similarity score.

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 1 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,

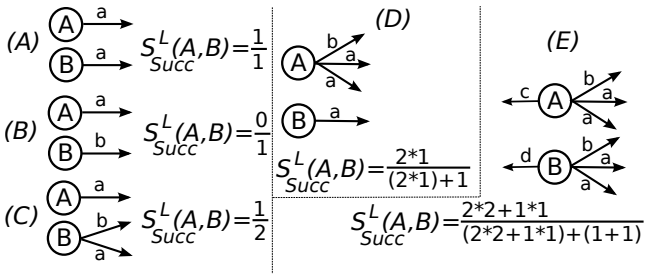


Figure 1. Non-recursive score computation

in Figure 1(D) it is impossible to say whether the transition $B \xrightarrow{a}$ matches 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. Givel PLTS X and 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:

$$Succ_{A,B} = \{(a, b, \sigma) \in Q_X \times Q_Y \times (\Sigma_X \cup \Sigma_Y) \mid A \xrightarrow{\sigma} a \wedge B \xrightarrow{\sigma} b\}$$

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 adjacent transitions. For outgoing transitions this is computed as follows (the L superscript stands for "local"):

$$S^L_{Succ}(A, B) = \frac{|Succ_{A,B}|}{|\Sigma_A - \Sigma_B| + |\Sigma_B - \Sigma_A| + |Succ_{A,B}|} \quad (1)$$

The equation above computes the set of all matching pairs of target states and transition labels with respect to outgoing transitions. However, 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:

$$Prev_{A,B} = \{(a, b, \sigma) \in Q_X \times Q_Y \times (\Sigma_X \cup \Sigma_Y) \mid a \xrightarrow{\sigma} A \wedge b \xrightarrow{\sigma} B\}$$

For incoming transitions, the score $Prev_{A,B}$ is defined in a similar way using *Prev* instead of *Succ* and $\{\sigma \in \Sigma | \exists t \in Q.t \xrightarrow{\sigma} s \in \Delta\}$ instead of Σ_s .

A pair of states (A, B) may be considered *incompatible*. This is the case if an outgoing transition $A \xrightarrow{x}$ is considered valid but $B \xrightarrow{x}$ is invalid (or vice versa). With respect to the PLTS, $x \in (\Sigma_A \cap \Psi_B) \cup (\Sigma_B \cap \Psi_A)$. If this is the case, $S(A, B) = -1$. By assigning negative scores, we are drawing a distinction between states that are merely dissimilar (which can in the worst case obtain a score of 0), and states that are definitely incompatible. It is important to note that the inclusion of incompatibility information is not crucial for the purposes of this work.

B. Scoring global similarity

The previous subsection shows how states are matched in terms of their adjacent transitions. However, we 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.

We start with the local similarity scoring algorithm for (potentially non-deterministic) states shown in the equation (1). The score is entirely dependent on the number of matching transitions $|Succ_{A,B}|$ and $|Prev_{A,B}|$. We extend this to aggregate the similarity score for every successive matched pair of transitions.

$$S^{G1}_{Succ}(A, B) = \frac{1}{2} \frac{\sum_{(a,b,\sigma) \in Succ_{A,B}} \left(1 + S^{G1}_{Succ}(a, b)\right)}{|\Sigma_A - \Sigma_B| + |\Sigma_B - \Sigma_A| + |Succ_{A,B}|}$$

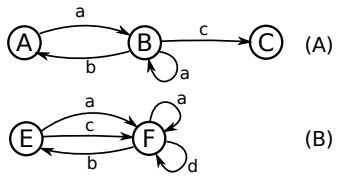
This equation augments the local similarity equation to recursively account for the similarity of next states. This can however lead to unintuitive scores because the score of every successive pair of states is given an equal precedence. The score can be skewed by high scores for state pairs that are 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 (2):

$$S^G_{Succ}(A, B) = \frac{1}{2} \frac{\sum_{(a,b,\sigma) \in Succ_{A,B}} \left(1 + k\, S^G_{Succ}(a, b)\right)}{|\Sigma_A - \Sigma_B| + |\Sigma_B - \Sigma_A| + |Succ_{A,B}|} \quad (2)$$

The fraction in front ensures that $|S^G_{Succ}(A, B)| \leq 1$. In a similar way, we may define $S^G_{Prev}(A, B)$ using *Prev* instead of *Succ*.

Given two PLTS X and Y, it is possible to solve the system of equations for every pair $(A, B) \in Q_X \times Q_Y$. The equation system is obtained by supplying the local knowledge for every pair of states (i.e. $Succ_{A,B}$ and $\Sigma_A - \Sigma_B$), so that the only unknown variables for each equation are the scores of the succeeding (or preceding) pairs of states.

Figure 2 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 obtained by rearranging the scoring equation $S^G_{Succ}(A, E)$ (equation (2)). 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 $S(A, E) = \frac{1}{2}\frac{1+kS(B,F)}{0+1+1}$. This 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 combined to form a score



	AE	AF	BE	BF	CE	CF	
AE	4			-k			1
AF	-k	6		-k			1
BE			6	-k		-k	2
BF	-k			8-k			2
CE					1		0
CF						1	0

Figure 2. Illustration of a system of equations for computation of scores in the general case.

as shown in equation (3):

$$S(A,B) = \begin{cases} \frac{S^G_{Succ}(A,B)+S^G_{Prev}(A,B)}{2} & \text{compatible} \\ -1 & \text{incompatible} \end{cases} \quad (3)$$

IV. The PLTSDiff Algorithm

Our PLTSDiff 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' [8] – 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.

A. Identifying Key Pairs (Landmarks)

Section III shows how to compute the similarity of two states. The pairs of states with the highest scores are chosen to be *key pairs*, and are the starting point for the PLTSDiff 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, and consider only those pairs that fall above that parameter (e.g. consider only the top 25%). However, a state may happen to be similar to several other states; even if it is well matched to them, it is unclear which of those states it should be paired with. For this reason, the second criterion is introduced: a ratio of the best match to the second best score. With such a ratio, only pairs where the best match is at least twice as good as any other match are added to the set of key pairs.

B. Computing the difference between machines

The process of comparing two graphs 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 graphs is similar – easily identifiable states (e.g. states with a unique surrounding topology of states and transitions) are used as landmarks, and the rest of the comparison is carried out with respect to them.

Input: PLTS_X, PLTS_Y, t, k
/* PLTSs are the two machines, t is the threshold-ratio pair that is used to identify key pairs, and k is the attenuation value */
Data: *KPairs*, *PairsToScores*, *NPairs*
Result: (*Added*,*Removed*,*Renamed*)
/* two sets of transitions and a relabelling */
1 *PairsToScores* $\leftarrow$ *computeScores*($\text{PLTS}_X, \text{PLTS}_Y, k$);
2 *KPairs* $\leftarrow$ *identifyLandmarks*(*PairsToScores*, t);
3 **if** *KPairs* $= \emptyset$ *and* $S(p_0, q_0) \geq 0$ **then**
4 *KPairs* $\leftarrow (p_0, q_0)$;
/* p_0 is the initial state in PLTS_X, q_0 is the initial state in PLTS_Y */
5 **end**
6 *NPairs* $\leftarrow \bigcup_{(A,B)\in \textit{KPairs}}$ *Surr*(A,B) $-$ *KPairs*;
7 **while** *NPairs* $\neq \emptyset$ **do**
8 **while** *NPairs* $\neq \emptyset$ **do**
9 $(A,B) \leftarrow$ *pickHighest*(*NPairs*, *PairsToScores*);
10 *KPairs* $\leftarrow$ *KPairs* $\cup (A,B)$;
11 *NPairs* $\leftarrow$ *removeConflicts*(*NPairs*, (A,B));
12 **end**
13 *NPairs* $\leftarrow \bigcup_{(A,B)\in \textit{KPairs}}$ *Surr*(A,B) $-$ *KPairs*;
14 **end**
15 *Added* $\leftarrow \{B_1 \xrightarrow{\sigma} B_2 \in \Delta_Y \mid \nexists\big(A_1 \xrightarrow{\sigma} A_2 \in \Delta_X \wedge (A_1,B_1) \in \textit{KPairs} \wedge (A_2,B_2) \in \textit{KPairs}\big)\}$;
16 *Removed* $\leftarrow \{A_1 \xrightarrow{\sigma} A_2 \in \Delta_X \mid \nexists\big(B_1 \xrightarrow{\sigma} B_2 \in \Delta_Y \wedge (A_1,B_1) \in \textit{KPairs} \wedge (A_2,B_2) \in \textit{KPairs}\big)\}$;
17 *Renamed* $\leftarrow$ *KPairs*;
18 **return** *(Added, Removed,Renamed)*

Algorithm 1: PLTSDiff

The PLTSDiff 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, 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 graph areas, until no further pairs of features can be found. What is left is the difference between the two machines. A set of key-pairs *KPairs* is a set of matching features of landscape, initialised to pairs of

landmarks and iteratively extended. To simplify the process of matching, we assume that *KPairs* is a one-to-one function; this property is enforced by the construction of the initial set of landmarks and subsequently preserved when this set is extended.

The algorithm is shown in algorithm 1. The functions *computeScores*$(\text{PLTS}_X, \text{PLTS}_Y, k)$ and *identifyLandmarks*$(PairsToScores, t)$ were described in the previous section; $Surr_{A,B} = \{(a,b) \in Q_X \times Q_Y \mid \exists \sigma \in (\Sigma_X \cup \Sigma_Y).\big((A \xrightarrow{\sigma} a \wedge B \xrightarrow{\sigma} b) \vee (a \xrightarrow{\sigma} A \wedge b \xrightarrow{\sigma} B)\big)\}$. Having started with landscapes, lines 6 and 13 compute candidate landmarks in order to extend *KPairs*, by considering not only transitions with the same label in the forward direction but also in the inverse one (hence *Surr*(A,B) is used). The loop in lines 8-11 uses candidates from *NPairs* to extend *KPairs* (this corresponds to a navigator looking around known features so as to match new ones). The set of pairs *NPairs* may contain numerous pairs, inclusion of which will violate the one-to-one property, this is why the best pair is chosen first (*pickHighest*(*NPairs*, *PairsToScores*)) and then all pairs which share either the first or the second element with the chosen one are eliminated from *NPairs* (using *removeConflicts*$(P,(A,B)) = \{(a,b) \in P \mid a \neq A \wedge b \neq B\}$). Subsequently, the next pair is chosen and so on until no pairs are left in *NPairs*. The extension of the boundary of exploration is performed by the loop in lines 7-14 where the algorithm alternates between the computation of a new set of candidates and choosing what to add to the set of key pairs.

Lines 15-17 guarantee that regardless of the choices made earlier in the algorithm the triple (*Added*, *Removed*, *Renamed*) forms a valid patch, that is, when transitions in *Removed* are removed from the first machine, then relabelling *Renamed* is applied to the remaining vertices and finally transitions in *Added* are added, the outcome has the same transitions as those of the second machine.

Theorem 4.1 (Correctness of patch generation): Assume that (1) names of states in PLTSs X and Y do not intersect (this can be accomplished using a suitable renaming) and consider PLTS BIG consisting of all states in Q_X and Q_Y. We additionally assume that (2) before a patch is applied, $\Delta_{BIG} = \Delta_X$ and (3) *Renamed* : $Q_X \to Q_Y$ is a one-to-one function. If (1)-(3) are satisfied, the application of the patch computed by algorithm 1 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 $\{A_1 \xrightarrow{\sigma} A_2 \mid \exists A_1, A_2 \in Q_X; B_1, B_2 \in Q_Y.\big((A_1,B_1) \in$ *KPairs* $\wedge\, (A_2,B_2) \in$ *KPairs* $\wedge\, A_1 \xrightarrow{\sigma} A_2 \in \Delta_X \wedge B_1 \xrightarrow{\sigma} B_2 \in \Delta_Y\big)\}$, thus 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*. 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 PLTS B. ■

Since multiple key pairs may have the same score, *pickHighest* is non-deterministic, although in practice it would use secondary measures such as state identifiers to return the same result for the same set of pairs.

Theorem 4.2 (Symmetry of the patch): Given PLTS X and Y, PLTSDiff(X,Y) can return (*Added*, *Removed*, *Renamed*) if and only if PLTSDiff(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 PLTSDiff(Y,X) computes *Renamed*$^{-1}$, with this in place the symmetry follows from the definitions of *Removed* and *Added*.

In order to demonstrate that *Renamed* from PLTSDiff(X,Y) is the same as *Renamed*$^{-1}$ from PLTSDiff(Y,X), we have to prove that construction of *KPairs* is symmetrical. This follows from (1) computation of scores $S(X,Y)$ is symmetric, hence *PairsToScores* computed by *computeScores* is symmetric; (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 PLTSDiff(X,Y) is the same as *KPairs*$^{-1}$ of PLTSDiff(Y,X). ■

The algorithm described above does not directly address vertices which are not connected anywhere. For a PLTS G, these can be defined by $UC_G = \{q \in Q_G \mid (\nexists A \in Q_G, \sigma \in \Sigma_G.q \xrightarrow{\sigma} A) \wedge (\nexists A \in Q_G, \sigma \in \Sigma_G.A \xrightarrow{\sigma} q)\}$. Since such vertices have no transitions leading to or leaving them, many of them can be accounted for by adding pairs of them to *Renamed*, with the rest handled by introducing *AddedUV* and *RemovedUV* to the patch. With an arbitrary renaming from UC_X to UC_Y included in $Renamed$, *RemovedUV* $= \{A \in UC_X \mid \nexists B \in UC_Y.(A,B) \in$ *Renamed* $\}$ and *AddedUV* $= \{B \in UC_Y \mid \nexists A \in UC_X.(A,B) \in$ *Renamed* $\}$.

C. Computing Structural Precision and Recall

PLTSDiff returns the structural difference between two FSMs. For certain tasks, such as comparing the accuracy of two inferred state machines, it is however also necessary to

make some quantitative evaluation of their accuracy. Given the output from PLTSDiff, this is relatively straightforward. Here we show how the output can be used to compute the Precision and Recall [3] of a state machine. Previous work by the authors [2] shows how Precision and Recall can be computed with respect to the languages of two machines; this section provides an equivalent measure with respect to the actual state transition structure.

Precision and recall is a measure that was originally developed by van Rijsbergen to evaluate the accuracy of information retrieval techniques [3]. The measure consists of two scores: Precision measures the 'exactness' of the subject, and recall measures the 'completeness'. Given a set REL of relevant elements, and a set RET of returned items, precision and recall are computed as follows: $Precision = \frac{|REL \cap RET|}{|RET|}$, $Recall = \frac{|REL \cap RET|}{|REL|}$.

In our case, given two state machines X and Y, we assume that machine X is considered 'relevant'. Thus, $rel = \Delta_X$ and $ret = \Delta_Y$. The intersection of ret and rel is computed with the help of PLTSDiff, and can be defined as follows: $REL \cap RET = \Delta_X - \textit{Removed}$.

D. Implementation

The PLTSDiff algorithm has been implemented as part of our state machine inference and analysis framework StateChum (http://statechum.sourceforge.net). 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 [9], 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 [10], which can be calibrated to take advantage of any specific hardware features such as multicore processors and cache size. The most computationally-intensive part of PLTSDiff is the construction and solution of a system of linear equations. For this reason, in a preliminary study to assess the scalability of PLTSDiff, pairwise scores have been computed between states of a few random state machines of various sizes. Such a computation proved feasible for machines with up to ≈500 states and 1-2 transitions from each state using a dual-core 2.4Ghz CoreDuo2™ with 3GB of memory running 64-bit Linux. With more states or greater density, computation of scores may run out of memory. For this reason, one may simply start with the pair of initial states as the only key pair before running the rest of PLTSDiff algorithm.

V. Practical Example and Discussion

This section serves to show how PLTSDiff can be used in practice. This is illustrated with a small example of how to compare the state machines that are produced by two diverse state machine inference techniques. This is followed by a discussion of the merits and potential problems that can arise when applied in practice.

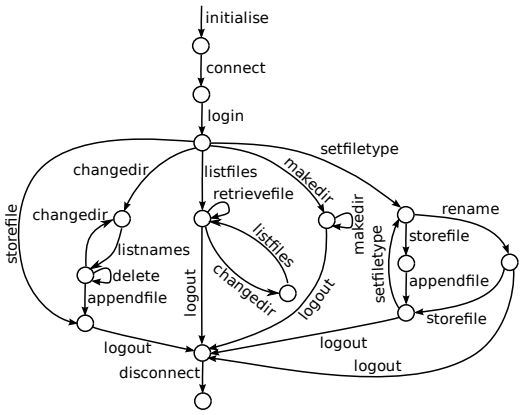


Figure 3. The original specification of the CVS client

There are a host of scenarios, especially in the domain of model-driven development, that rely on the comparison of different models. In their work on comparing Statechart specifications for example, Nejati *et al.* [11] motivate their comparison approach by comparing two Statecharts of different versions of a call-logging system (one with and one without voicemail). The development of our comparison technique was largely spurred by the need to be able to evaluate the accuracy of reverse engineered models [12], [2] in terms of their structure.

Conventionally, evaluations of reverse-engineering techniques neglect the structure of the models and concentrate solely on the external behaviour (the languages) [13], [14]. Here, we show how the PLTSDiff algorithm can be used to provide a more insightful, structural comparison that can be used to complement the language-based measures. As a basis we will use a small model of a CVS client (shown in figure 3 – derived from a similar model by Lo *et al.* [14]). A random sample of traces is taken from this model and these are used as input for two different inference techniques.

The two techniques we use are Markov and EDSM. Markov was proposed by Cook and Wolf [15] and uses the concept of Markov models to find the most probable state machine. EDSM [16] is an inference algorithm that originated in the domain of grammar-inference. Without going into detail, EDSM is a heuristic alternative to the k-Tails algorithm [17] that scores sets of state-pairs before merging the pair with the highest score. To save space, the individual reverse-engineered models are not included, but can be downloaded[1] (they are part of the diff- models shown in figures 4 and figure 5 anyway).

[1] http://www.dcs.shef.ac.uk/~nw/Files/wcre/edsm2.pdf
http://www.dcs.shef.ac.uk/~nw/Files/wcre/markov.pdf

	$Prec_{Lang}$	Rec_{Lang}	$Prec_{FSM}$	Rec_{FSM}
Markov	1	0.45	0.60	0.44
EDSM	0.47	0.8	0.42	0.37

Table I
PRECISION AND RECALL RESULTS FOR LANGUAGE AND STRUCTURE

A. Results

A conventional evaluation would compare the reverse-engineered models to the reference model in figure 3 in terms of their languages [14], [2]. In comparing the languages, the aim is to identify the extent to which the sequences of events that are produced by one model overlap with the sequences of events that are produced by the other. A precision and recall-based comparison of the languages [2] is provided in table I under headings $Prec_{Lang}$ and Rec_{Lang}. Precision measures what proportion of valid event sequences in the inferred model are present in the target. Recall measures the proportion of valid event sequences in the target that are captured by the inferred model. It is possible to also compute "negative" precision and recall to measure the accuracy of the machines with respect to *invalid* sequences of events, but we omit these here for the sake of simplicity.

The language measures suggest that the model produced by the Markov learner is more precise than EDSM. In fact, it suggests that the language of the Markov model is a proper subset of the language of the target model. The EDSM model has a much higher recall, which means it correctly inferred a broader range of target language. However, the low precision score suggests that it includes a lot of false positive sequences. This is about as much as can be ascertained from language-based scores; the generality of one machine versus an other. We have no idea of which particular areas of the machine have been correctly inferred or missed out.

By applying the PLTSDiff algorithm, we can gain more concrete insights comparing the essential components of the inferred machines; their states and transitions. The output from the PLTSDiff algorithm for the markov model is presented in figure 4 and figure 5 for the EDSM model. 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 PLTSDiff 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 precision and recall measures introduced in section IV-C are presented in table I under headings $Prec_{FSM}$ and Rec_{FSM}. These present a completely different perspective of the accuracy which, along with the language-based scores can be used to provide a more complete picture. Scores still show that the Markov model is more precise, but the fact that it contains superfluous edges prevents it from achieving a perfect score. In terms of recall, the EDSM is missing more transitions than the Markov model. EDSM can often produce models that are too general in terms of their language, because they tend to merge too many states together. Although these merges are often wrong, the language-based score rewards this by a higher recall. However, under the structural precision and recall scores produced here, this is penalised, by accounting for the set of transitions that disappear as a result of erroneous mergers. This helps to provide a more balanced assessment of the accuracy of the final model.

Importantly, the structural results permit us to see exactly which transitions were (in-)correctly inferred by the different techniques. Thus we establish that there is very little overlap between the two inferred machines. 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.

B. Discussion

PLTSDiff has a number of advantages over existing state machine comparison techniques. Most state machine comparison techniques require their inputs to be minimal and deterministic, with all states reachable from the initial state. This is not necessary for PLTSDiff, which can just as easily be applied to non-minimal and non-deterministic machines. The Markov model above is non-deterministic. It also demonstrates that PLTSDiff can accept graphs as input that are not even fully connected.

Besides its versatility, the technique also tends to be very good at optimising the difference between two automata. PLTSDiff is likely to identify the matching state-pairs, whereas other techniques are not always as likely to do so (see section VI). Given that the landmark pairs are always likely to be correctly matched, it will usually produce the smallest *Added* and *Removed* sets.

Whenever PLTSDiff performs poorly, most transitions of X are deemed removed and all of Y – added. This may happen if few key pairs are found or when *identifyLandmarks* chooses too many pairs, many of them spurious. In the former case, this would be because the threshold-ratio is too 'strict' and allowing too few pairs to be chosen. On graphs with disconnected parts, some of these parts may have no state chosen for a key pair, leading the entire part to feature in the *Removed* part and the corresponding part

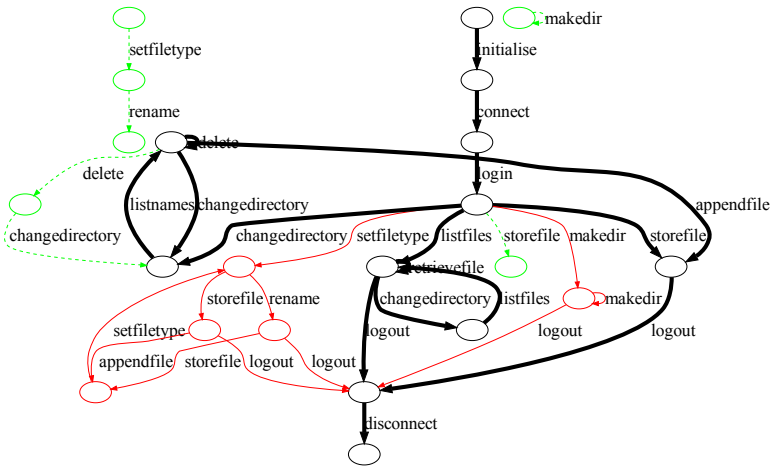


Figure 4. Results of PLTSDiff for Markov model

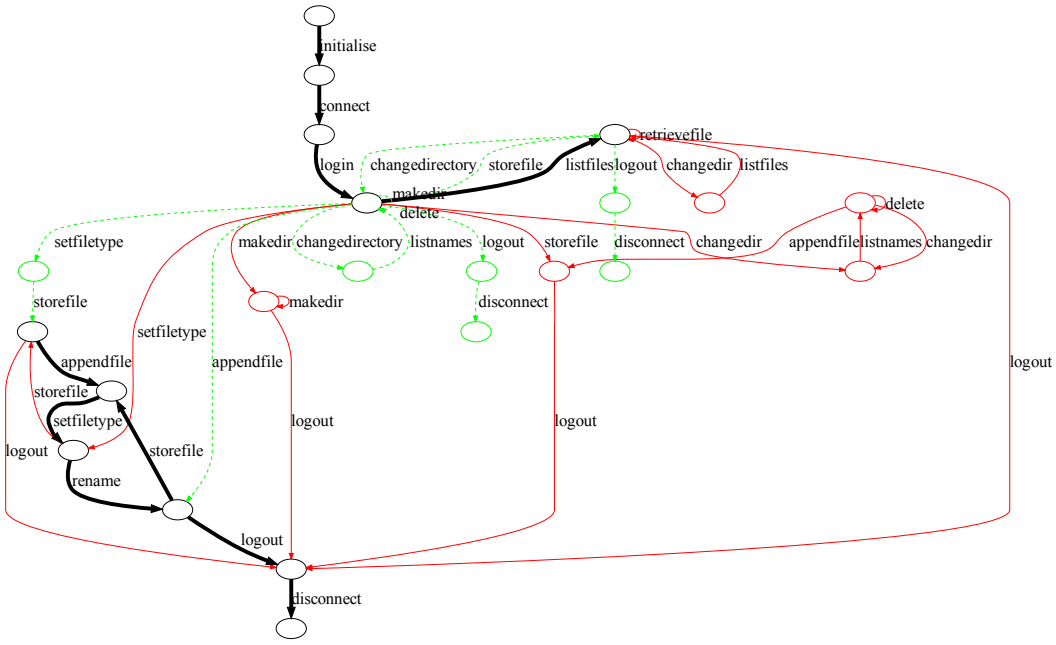


Figure 5. Results of PLTSDiff for EDSM model

from the second graph — in the *Added* one. When too many pairs are chosen, many of them will have few 'matched' transitions, with all the remaining ones featuring in *Removed* and *Added*. Such a poor performance can be exhibited even when matching a graph with itself and can be dealt with by adjusing t and k parameters in algorithm 1.

The selection of the values of t and k depends 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. When this is the case, anecdotal experience by the authors suggests that setting the threshold value to top 25%, ratio of the best match to the second best to 2, and the attenuation factor to 0.5 produces correct results.

VI. Related Work

Nejati *et al.* have proposed a similar structural comparison technique as part of a broader approach to comparing UML Statecharts [11]. Their technique relies on a machine where all states can be reached from the initial node, and that 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. There are two potential problems here; first, the similarity scores for a given pair of states can only be valid if the pairs of states that were identified in the prior run-up are themselves equivalent. In the scenario where segments of the machine that may be far removed from the initial node are similar, but the rest of the machines are completely different, this approach may produce unreliable results. The second problem is that the fixed point may not be reached - meaning that the computation may have to be stopped at an arbitrary point. As part of their broader Statechart-based approach, Nejati *et al.* adopt some useful techniques for comparing the attribute labels of states and transitions, to increase the confidence that are a pair of states are equivalent; although our technique only considers LTS's, their complementary techniques could prove useful in the presence of additional state / transition labels.

The technique presented in this paper offers some important improvements. It is important to note that it solely concentrates on comparing transition structure, and does not consider other complementary measures also considered by Nejati *et al.* (however several of their other measures could easily be incorporated into our approach). The use of a linear equation solver ensures that computation of a score for a pair is not dependent on a specific path used to reach the pair from the initial state and fixpoint is always reached. Thus, even in cases where small isolated areas of the machine are very similar to their counterparts in the other machine, these will be identified. Finally, PLTSDiff starts with a carefully chosen set of pairs and propagates them in all directions using precomputed scores while [11] considers the two directions completely separately.

Fixed-point computation for state matching is also utilised in [18] where (1) the notion of state-dependent 'propagation coefficient' is used – we use the same value of attenuation factor and (2) filtering of results of matching uses a number of different techniques in order to approximate results obtained by a human. We plan to consider filtering techniques of [18] in our future work.

In their work on evaluating reverse-engineered automata, Quante and Koschke [19] present a technique with a similar aim to ours. Their approach works by computing the union of the two machines [20], 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 original machine. 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 [21], aimed at finding an alignment of proteins maximising their overlapping segments. Both tasks involve 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 [22]; 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 [22] 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 [22] 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 PLTSDiff considers directed graphs means that (1) the outcome of a recursive score computation is of good quality and (2) PLTSDiff can use these scores and avoid backtracking.

The challenge is a specific instance of the graph isomorphism problem, called error-correcting graph isomorphism [23]. 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 [24] describes an inference process, using a procedure similar 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 [24] determines when two states can be considered probabilistically behaviourally-equivalent.

VII. Conclusions and Future Work

We have presented an algorithm for computing the difference between two finite state machines, along with a proof-of-concept implementation. The PLTSDiff algorithm has several advantages over existing approaches to state machine comparison. Machines can be non-deterministic, non-minimal, and not connected. The latter means that a machine may contain groups of states with no transitions between these groups (three groups in Figure 4). The implementation of PLTSDiff has been tested on reasonably large state machines, and appeared to scale well for state machines with hundreds of states.

In our future work we intend to carry out a more formal study into the scalability of the approach. The technique depends on solving linear systems of equations. This is known to scale reasonably well if the matrix of equations is sparse, and it will be our priority to investigate how the scale

is effected by dense state machines, which would produce matrices that are also correspondingly dense.

ACKNOWLEDGEMENTS

This research is supported by the EPSRC-funded AutoAbstract and REGI grants (EP/C511883/1, EP/F065825/1) and the EU FP7 ProTest project. The authors thank Jonathan Cook at New Mexico State University for supplying the Balboa software, which contained the Markov learner in the case study.

REFERENCES

[1] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin, "Improving dynamic software analysis by applying grammar inference principles," *Journal of Software Maintenance and Evolution: Research and Practice*, 2008.

[2] N. Walkinshaw, K. Bogdanov, and K. Johnson, "Evaluation and comparison of inferred regular grammars," in *Proceedings of the International Colloquium on Grammar Inference (ICGI'08)*, St. Malo, France, 2008.

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

[4] E. Börger, "Abstract state machines and high-level system design and analysis," *Theoretical Computer Science*, vol. 336, no. 2-3, pp. 205–207, 2005. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2004.11.006

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

[6] S. Uchitel, J. Kramer, and J. Magee, "Behaviour model elaboration using partial labelled transition systems," in *ESEC/SIGSOFT FSE*. ACM, 2003, pp. 19–27. [Online]. Available: http://doi.acm.org/10.1145/940071.940076

[7] V. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals." *Soviet Physics Doklady.*, vol. 10, no. 8, pp. 707–710, Feb. 1966, doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[8] M. Sorrows and S. Hirtle, "The nature of landmarks for real and electronic spaces," in *Spatial information theory - Cognitive and computational foundations of geographic information science*. Berlin: Springer, 1999, pp. 37–50.

[9] T. Davis, "Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method," *ACM Trans. Math. Softw.*, vol. 30, no. 2, pp. 196–199, 2004.

[10] R. Whaley and A. Petitet, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, February 2005.

[11] S. Nejati, M. Sabetzadeh, M. Chechik, S. M. Easterbrook, and P. Zave, "Matching and merging of statecharts specifications," in *ICSE*. IEEE Computer Society, 2007, pp. 54–64. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ICSE.2007.50

[12] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin, "Reverse engineering state machines by interactive grammar inference," in *WCRE'07*, 2007.

[13] N. Walkinshaw and K. Bogdanov, "Inferring finite-state models with temporal constraints," in *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE'08)*, 2008.

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

[15] J. Cook and A. Wolf, "Discovering models of software processes from event-based data," *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 3, pp. 215–249, 1998.

[16] K. Lang, B. Pearlmutter, and R. Price, "Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm," in *Proceedings of the International Colloquium on Grammar Inference (ICGI'98)*, vol. 1433, 1998, pp. 1–12.

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

[18] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity flooding: A versatile graph matching algorithm and its application to schema matching," in *18th International Conference on Data Engineering (ICDE 2002)*, 2002. [Online]. Available: http://ilpubs.stanford.edu:8090/730/

[19] J. Quante and R. Koschke, "Dynamic protocol recovery," in *Proceedings of the International Working Conference on Reverse Engineering (WCRE'07)*, 2007, pp. 219–228. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/WCRE.2007.24

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

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

[22] J. Raymond, E. Gardiner, and P. Willett, "RASCAL: Calculation of graph similarity using maximum common edge subgraphs," *The Computer Journal*, vol. 45, no. 6, pp. 631–644, 2002.

[23] H. Bunke, "On A relation between graph edit distance and maximum common subgraph," *Pattern Recognition Letters*, vol. 18, no. 8, pp. 689–694, Aug. 1997.

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