A Quasi-local Algorithm for Checking Bisimilarity

Wenjie Du
Shanghai Normal University, China

Yuxin Deng
Shanghai Jiao Tong University, China
Zhejiang Normal University, China

***Abstract*—Bisimilarity is one of the most important relations for comparing the behaviour of formal systems in concurrency theory. Decision algorithms for bisimilarity in finite state systems are usually classified into two kinds: global algorithms are generally efficient but require to generate the whole state spaces in advance, local algorithms combine the verification of a system's behaviour with the generation of the system's state space, which is often more effective to determine that one system fails to be related to another. In this paper we propose a quasi-local algorithm with worst case time complexity $O(m_1 m_2)$, where m_1 and m_2 are the numbers of transitions in two labelled transition systems. With mild modifications, the algorithm can be easily adapted to decide similarity with the same time complexity. For deterministic systems, the algorithm can be simplified and runs in time $O(\min(m_1, m_2))$.**

***Index Terms*—Concurrency; labelled transition systems; bisimilarity; algorithm**

I. Introduction

In the last three decades a wealth of behavioural equivalences have been proposed in concurrency theory. Among them, *bisimilarity* [12], [14] is probably the most studied one as it admits a suitable semantics and an elegant co-inductive proof technique. It can also be given an efficient decision procedure [13]. Given an labelled transition system (LTS) with n states and m transitions, the partition refinement algorithm of Paige and Tarjan takes time $O(m \log n)$ to generate all bisimulation equivalence classes. Although this algorithm is efficient, it requires to generate the whole state space of a system in advance. However, in many cases, one may be able to determine that one process fails to be related to another by examining only a fraction of the state space. One would like to have a verification algorithm that exploits this fact. "On the fly" algorithms combine the verification of a system's behaviour with the generation of the system's state space.

Fernandez and Mounier [5] first proposed an "on the fly" algorithm for checking behavioural equivalences and preorders. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two LTSs with initial states s_0 and t_0. To decide if s_0 is bisimilar to t_0, the algorithm of [5] performs depth-first searches (DFS for short) on the product LTS $\mathcal{L}_1||\mathcal{L}_2$. During a round of DFS, it is possible to reach a state, say $(s||t)$, which has already been visited because of a loop, but not yet analyzed. In this case, s and t are *assumed* to be bisimilar and the DFS continues. If the two states are found to be not bisimilar after finishing searching the loop, then we know that a wrong assumption was used. So another round of DFS has to be performed, with one piece of new information, namely the two states are not bisimilar. In the worst case, each round of DFS only yields one new pair of states that are not bisimilar, and most of the time it repeats checking the states that are already considered in the previous round of DFS. This observation leads us to study an improved algorithm that examines each pair of states at most once.

The basic idea for our algorithm of checking bisimilarity is as follows. We start with constructing the product of $\mathcal{L}_1$ and $\mathcal{L}_2$ and marking state $(s||t)$ in $\mathcal{L}_1||\mathcal{L}_2$ with label 0 if s and t have different initial actions. All other states are given label 1. Then we propagate label 0 step by step to all pairs of states that are not bisimilar. The propagation of 0-labels proceeds in a backward way. For example, if $(s||t) \xrightarrow{a} (s'||t')$ is the unique outgoing transition from $(s||t)$ with label a and $(s'||t')$ has been labelled by 0, then $(s||t)$ will be labelled by 0 as well. That is, the label 0 will be propagated from $(s'||t')$ to its predecessor $(s||t)$. The intuition is that if s' is not bisimilar to t' then s is not bisimilar to t either because the two transitions $s \xrightarrow{a} s'$ and $t \xrightarrow{a} t'$ cannot match each other. In general, if $(s||t)$ has several a-labelled outgoing transitions, the situation is more complicated, and we need to use some particular data structures to implement the propagation of 0-labels. Eventually the procedure of label propagation stops when (i) either the initial state $(s_0||t_0)$ is reached, (ii) or we end up in a stable situation without reaching $(s_0||t_0)$ because no state can propagate its 0-label to any of its predecessors. In case (i) $(s_0||t_0)$ will be given label 0 to indicate that s_0 is not bisimilar to t_0; in case (ii) $(s_0||t_0)$ keeps its label 1 to indicate that s_0 is bisimilar to t_0. It turns out that in a stable situation state $(s||t)$ has label 0 if and only if s is not bisimilar to t. This property ensures the correctness of our algorithm. Moreover, the basic idea illustrated above is also applicable to checking similarity.

Let n_1 and m_1 (resp. n_2 and m_2) be the numbers

of states and transitions in $\mathcal{L}_1$ (resp. $\mathcal{L}_2$), respectively. Our algorithm always terminates within time $O(m_1 m_2)$, while the algorithm in [5] requires time $O(n_1^2 n_2^2)$ in the worst case. On the other hand, the space requirement is $O(m_1 m_2 + n_1^2 n_2 + n_1 n_2^2)$ for our algorithm, and $O(m_1 m_2)$ for the algorithm in [5]. The reason is that our algorithm is *quasi-local* in the sense that the synchronous product $\mathcal{L}_1 || \mathcal{L}_2$ is generated by a preprocessing step, while in a local algorithm the product is dynamically generated, which trades time for space.

In the particular case of verifying bisimilarity and similarity on deterministic LTSs, our algorithm can be greatly simplified, with both time and space complexities being $O(\min(m_1, m_2))$.

Related work. There are mainly two kinds of algorithms for checking behavioural equivalences, as classified in [3]. *Global* algorithms (e.g. [4], [6]) are based on the partition refinement technique and compute the equivalence classes of states after an LTS is fully generated; *local* algorithms (e.g. [5], [2], [10], [11]), also called "on the fly" algorithms, avoid the construction of unnecessary states by searching on the synchronous product of two LTSs for inequivalent states. Moreover, local algorithms can also be used for checking behavioural preorders, but global ones cannot. For local algorithms, besides the work [5] mentioned above, Lin [10] lifted Fernandez and Mounier's algorithm to handle value-passing processes. Celikkan [2] proposed a different preorder-checking algorithm by recursively constructing a graph whose vertices are pairs of related states. It takes time $O((n+m)^2)$ to check if two given states in an LTS are related by a behavioural preorder in the worst case. Recently, Mateescu and Oudot [11] proposed an algorithm by first encoding a bisimulation relation as a boolean equation system (BES) and then employing a local BES resolution algorithm. Given two LTSs $\mathcal{L}_1$ and $\mathcal{L}_2$, the generation of a BES takes time $O(m_1 m_2)$. Once the BES is constructed, which is represented as a boolean graph with the size of $\mathcal{L}_1 || \mathcal{L}_2$, say with n states and m transitions, the local BES resolution algorithm takes time $O(n+m)$ in the best case and $O((n+m)^2)$ in the worst case.

Our quasi-local algorithm somehow sits in the middle between global and local algorithms. Unlike local algorithms, our construction of the synchronous product of two LTSs is not "on the fly", but is done once and for all; unlike global algorithms, we do not search on the union of the state spaces of two LTSs, but on the state space of their synchronous product, and the latter is often smaller than the former, especially for deterministic systems, as discussed in Section IV.

Another approach to checking bisimilarity between two states is to find a *characteristic formula* [16] for one state in the modal μ-calculus [9] and check if the other state satisfies the formula. The time requirement for checking if two LTSs are related by bisimilarity is $O((n_1+m_1)(n_2+m_2))$. Moreover, checking behavioural equivalences can also be formulated as satisfiability problems for Horn clauses whose solutions can be found by using known algorithms [15].

Structure of the paper. In Section II we briefly introduce some basic concepts about bisimulation and its characterisation as infinite approximations. In Section III we present our algorithm for checking bisimilarity and similarity on finite state LTSs. In Section IV we see that for deterministic systems the algorithm can be greatly simplified. Finally, we give some concluding remarks in Section V.

II. Preliminaries

In this section we recall a few basic concepts and properties about bisimulation and simulation.

Definition 2.1: A *doubly labelled transition system (DLTS)* is a tuple $(S, L, A, \longrightarrow, s_0)$ where

- S is set of states
- $L : S \to \{0, 1\}$ is a labelling function
- A is a set of actions
- $\longrightarrow \subseteq S \times A \times S$ is a labelled transition relation
- s_0 is the initial state.

If we omit L, then states are not labelled but transitions are still labelled, so we obtain the usual *labelled transition systems (LTSs)*.

We will use the notation $s \xrightarrow{a} t$ to stand for $(s, a, t) \in \longrightarrow$. We shall only consider *simple LTSs*, which are LTSs with the constraint that if $s \xrightarrow{a} s'$ and $s \xrightarrow{b} s'$ then $a = b$. An LTS is said to be *deterministic* if $s \xrightarrow{a} s_1$ and $s \xrightarrow{a} s_2$ then $s_1 = s_2$, i.e. performing an action from a state will lead to a unique successor state. For $s \in S$, we define the set of states $Pred(s) = \{t \in S \mid \exists a \in A : t \xrightarrow{a} s\}$. We consider the set of initial actions that can be performed by a state s: $Init(s) = \{a \in A \mid \exists t \in S : s \xrightarrow{a} t\}$.

Definition 2.2: A binary relation R on the states of an LTS is a *simulation* if whenever $(s, t) \in R$:

- for all s' and a with $s \xrightarrow{a} s'$, there exists some t' such that $t \xrightarrow{a} t'$ and $(s', t') \in R$.

The relation R is a *bisimulation* if both R and R^{-1} are simulations. *Similarity* (resp. *Bisimilarity*), written $\preceq$ (resp. $\sim$), is the union of all simulations (resp. bisimulations).

Bisimilarity can be approximated by a family of inductively defined relations. Similarity can be approximated in a similar way.

Definition 2.3: Let S be the state set of an LTS. We define:

- $\sim_0 := S \times S$
- $s \sim_{n+1} t$, for $n \geq 0$, if
  1) for all s' and a with $s \xrightarrow{a} s'$, there exists some t' such that $t \xrightarrow{a} t'$ and $s' \sim_n t'$;
  2) for all t' and a with $t \xrightarrow{a} t'$, there exists some s' such that $s \xrightarrow{a} s'$ and $s' \sim_n t'$.
- $\sim_\omega := \bigcap_{n \geq 0} \sim_n$

In general, $\sim$ is a strictly finer relation than $\sim_\omega$. However, the two relations coincide when limited to *image-finite* LTSs, that is, for any state s the set of its derivatives $\{s' \mid s \xrightarrow{a} s', \text{ for some } a \in A\}$ is finite.

Proposition 2.4: On image-finite LTSs, $\sim_\omega$ coincides with $\sim$.

In the sequel, we consider *finitary* LTSs, which have finitely many states and transitions. Clearly, finitary LTSs are image-finite, which allows us to use the above proposition. We will use the symbol R to stand for both $\preceq$ and $\sim$.

III. Verifying bisimilarity and similarity

We consider two LTSs $\mathcal{L}_1 = (S, A_1, \longrightarrow_1, s_0)$ and $\mathcal{L}_2 = (T, A_2, \longrightarrow_2, t_0)$. The DLTS $\mathcal{L}_1||_R\mathcal{L}_2$ is defined via a synchronous product of $\mathcal{L}_1$ and $\mathcal{L}_2$, similar to the product given in [5]. States in $\mathcal{L}_1||_R\mathcal{L}_2$ are in the form $s||_R t$, but for simplicity we write $s||t$ instead. A state $(s||t)$ of $\mathcal{L}_1||_R\mathcal{L}_2$ can perform a transition labelled by action a if and only if s and t can perform a transition labelled by a in $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively. Otherwise,

- in the case of a simulation (R is $\preceq$), if some action can only be performed by s, i.e. $Init(s) \not\subseteq Init(t)$, then $(s||t)$ is a deadlock state labelled 0.
- in the case of a bisimulation (R is $\sim$), if some actions can be performed by only one of the two states (s or t), i.e. $Init(s) \neq Init(t)$, then $(s||t)$ is a deadlock state labelled 0.

Definition 3.1: We define the DLTS $\mathcal{L} = \mathcal{L}_1||_R\mathcal{L}_2$ to be the tuple $(Q, L, A_1 \cap A_2, \longrightarrow, (s_0||t_0))$, where Q, L and $\longrightarrow$ are the smallest sets obtained by the applications of the following rules:

1) $(s_0||t_0) \in Q$.
2) In the case that R is a simulation (resp. bisimulation), $L(s||t) = 1$ if $Init(s) \subseteq Init(t)$ (resp. $Init(s) = Init(t)$), otherwise $L(s||t) = 0$.
3) If $(s_1||t_1) \in Q$, $L(s_1||t_1) = 1$, $s_1 \xrightarrow{a}_1 s_2$ and $t_1 \xrightarrow{a}_2 t_2$ then $(s_2||t_2) \in Q$ and $(s_1||t_1) \xrightarrow{a} (s_2||t_2)$.

For simplicity, we sometimes omit the subscripts in $\longrightarrow_1$ and $\longrightarrow_2$.

Example 3.2: In Figure 1, diagrams (a) and (b) describe two LTSs with initial states s_0 and t_0, respectively. The DLTS obtained via their synchronous product, which is the same both for simulation and bisimulation, is described in (c), where the red color for state $(s_3||t_4)$ means that this state is labelled 0 because $Init(s_3) = \{b\}$ and $Init(t_4) = \{c\}$, while all other states are labelled 1 and have white color. □

Proposition 3.3: Let $(s||t)$ be a state in $\mathcal{L}_1||_R\mathcal{L}_2$. Then $(s,t) \notin R_k$ for a minimum number $k \geq 1$ if and only if there exists a sequence of states

$$(s_k||t_k), (s_{k-1}||t_{k-1}), ..., (s_1||t_1)$$

such that

1) $(s||t) = (s_k||t_k)$
2) $(s_{i+1}||t_{i+1}) \xrightarrow{a_i} (s_i||t_i)$ for all $i \in 1..(k-1)$ and some action a_i
3) $(s_i, t_i) \in R_{i-1}$ but $(s_i, t_i) \notin R_i$, for all $i \in 1..k$.

Proof: The "if" direction is straightforward. For the "only if" direction, we proceed by induction on k.

- In the base case $k = 1$, the sequence with only one state (s,t) satisfies our requirements.
- Suppose $k > 1$. Since $(s,t) \notin R_k$, without loss of generality we can assume the existence of some action a such that $s \xrightarrow{a} s_{k-1}$ is a transition which cannot be matched up by any transition from t. That is, for all $t \xrightarrow{a} t_i$ we have $(s_{k-1}, t_i) \notin R_{k-1}$. On the other hand, it follows from $(s,t) \in R_{k-1}$ that there is some t_{k-1} with $t \xrightarrow{a} t_{k-1}$ and $(s_{k-1}, t_{k-1}) \in R_{k-2}$. By induction hypothesis, there exists a sequence $(s_{k-1}||t_{k-1}), ..., (s_1||t_1)$ such that (1) $(s_i||t_i) \xrightarrow{a_i} (s_{i-1}||t_{i-1})$ for all $i \in 2..(k-1)$ and (2) $(s_i, t_i) \in R_{i-1}$ but $(s_i, t_i) \notin R_i$ for all $i \in 1..(k-1)$. Since $(s||t) \xrightarrow{a} (s_{k-1}||t_{k-1})$, the sequence $(s||t), (s_{k-1}||t_{k-1}), ..., (s_1||t_1)$ satisfies our requirements.

■

Corollary 3.4: Let $(s||t)$ be a state in $\mathcal{L}_1||_R\mathcal{L}_2$. Then $(s,t) \notin R$ if and only if there exists a sequence of states $(s_k||t_k), (s_{k-1}||t_{k-1}), ..., (s_1||t_1)$ for $k \geq 1$ such that

1) $(s||t) = (s_k||t_k)$
2) $(s_{i+1}||t_{i+1}) \xrightarrow{a_i} (s_i||t_i)$ for all $i \in 1..(k-1)$ and some action a_i
3) $(s_i, t_i) \in R_{i-1}$ but $(s_i, t_i) \notin R_i$, for all $i \in 1..k$.

Proof: Since $R = \bigcap_k R_k$ in our setting, it follows that $(s,t) \notin R$ if and only if there is a minimum natural number k such that $(s,t) \notin R_k$. Combining this with the above proposition yields the expected result. ■

Proposition 3.5: Let $\mathcal{L} = \mathcal{L}_1||_R\mathcal{L}_2$. If $L(s||t) = 0$ then $(s,t) \notin R$.

Proof: The result directly follows from Definition 3.1. ∎

In general, the converse of Proposition 3.5 does not necessarily hold. For example, in Figure 1 (c) state $(s_2||t_2)$ has label 1 but s_2 and t_2 are related neither by similarity nor bisimilarity. However, the initial DLTS generated by Definition 3.1 tells us those states that are obviously not related by R because one state can immediately enable an action that the other state cannot perform. In the next stage of our algorithm we will propagate the 0-labels so as to reach the initial state $(s_0||t_0)$ or a stable situation in which all states not related by R are labelled 0. The propagation of 0-labels is done in a backward way. We illustrate the basic idea by the example below.

Example 3.6: In the DLTS described by Figure 1 (c), the only state with label 0 is $(s_3||t_4)$. Clearly, $s_3 \not\preceq t_4$ because s_3 can exhibit action b while t_4 cannot. We now look at the state $(s_2||t_2)$ which is the unique predecessor of $(s_3||t_4)$. Here it happens that $(s_3||t_4)$ is the unique successor of $(s_2||t_2)$ with the first component being s_3. In other words, the underlying fact is that if the state s_2 in $\mathcal{L}_1$ makes the transition $s_2 \xrightarrow{b} s_3$, the only possibility for t_2 in $\mathcal{L}_2$ to match up is to make the transition $t_2 \xrightarrow{b} t_4$. As we already know that $s_3 \not\preceq t_4$, we can draw the conclusion that $s_2 \not\preceq t_2$ either. Therefore, the original label, which is 1, for the state $(s_2||t_2)$ is not accurate, and we should change it into 0, i.e. paint $(s_2||t_2)$ red. This observation leads us to let the DLTS in Figure 1 (c) evolve into the one in Figure 2 (a); the latter has one more 0-labelled state than the former. With a similar backward analysis we find that the state $(s_1||t_1)$, the only predecessor of $(s_2||t_2)$ should be relabelled 0, thus yielding Figure 2 (b). Repeating this procedure, which is convergent because the number of 1-labels is decreasing, and eventually we reach the stable situation in Figure 2 (c). Now $(s_0||t_0)$ is labelled 0, indicating that $s_0 \not\preceq t_0$. □

Remark 3.7: The idea illustrated in Example 3.6 is akin to an algorithm of minimising deterministic finite automata (DFA) [7], [8]. However, that algorithm cannot be directly transplanted to LTSs because of the presence of nondeterminism, i.e. a state may enable several outgoing transitions labelled with the same action. Roughly speaking, we generalise that algorithm to a setting with nondeterminism, and the generalisation employs the data structure of three-dimensional arrays, as we will see in the sequel.

Example 3.8: Suppose we would like to know if $t_0 \preceq s_0$ holds. The same initial DLTS in Figure 1 (c) can be used, but each state $(s||t)$ corresponds to a state $(t||s)$ in the above analysis. The state $(s_3||t_4)$ is still labelled 0, and indeed $t_4 \not\preceq s_3$ because the former can immediately enable action c while the latter cannot. As before we try to propagate label 0 to $(s_2||t_2)$, the unique predecessor of $(s_3||t_4)$. However, besides $(s_3||t_4)$ the state $(s_2||t_2)$ has another successor $(s_4||t_4)$. In other words, the transition $t_2 \xrightarrow{b} t_4$ could possibly be matched up by any of the two outgoing transitions from s_2: $s_2 \xrightarrow{b} s_3$ and $s_2 \xrightarrow{b} s_4$. In this case, we cannot simply propagate a 0-label from $(s_3||t_4)$ to $(s_2||t_2)$. The failure of s_3 to simulate t_4 does not necessarily leads to a failure of s_2 to simulate t_2, since s_2 has an alternative transition $s_2 \xrightarrow{b} s_4$ which might be able to match up $t_2 \xrightarrow{b} t_4$ successfully. As a matter of fact, the alternative transition does indeed allow s_2 to simulate t_2 successfully because $t_4 \preceq s_4$ holds. So our attempt of propagating 0-label to $(s_2||t_2)$ stops, i.e. the situation in Figure 1 (c) is already stable. Now $(s_0||t_0)$ is labelled 1, indicating that $t_0 \preceq s_0$ □

In general, the propagation of labels is more complicated than what we have seen in Example 3.6, as alluded in Example 3.8. The complication is mainly entailed by the following two factors.

1) Usually there are more than one 0-labelled states. From any of them we can start a 0-label propagating procedure, though the rate of reaching a stable situation might be different.
2) Usually a state has more than one successors and more than one predecessors. It might be possible to propagate the 0-label from a state to some of its predecessors, while some other predecessors might not change their labels because of their alternative transitions as we have seen in Example 3.8.

To address the two issues above, we introduce in the algorithm **CheckBisim** (cf. Algorithm 1) the following data structures.

- A stack St stores 0-labelled states. As long as St is not empty, we remove the top element of the stack and start a round of 0-label propagation from it. Each time we relabel a state by 0 we also push it into St.
- A three-dimensional array of integers $Ar_1[1..n_1, 1..n_1, 1..n_2]$ in the case of a simulation, and also a three-dimensional array $Ar_2[1..n_1, 1..n_2, 1..n_2]$ in the case of a bisimulation. Since $\mathcal{L}_1$ is assumed to be a simple directed graph, for any $k, i \in 1..n_1$ there is at most one action a with the transition $s_k \xrightarrow{a} s_i$. This transition, if it exists, can possibly be matched up by some candidate transitions in $\mathcal{L}_2$.

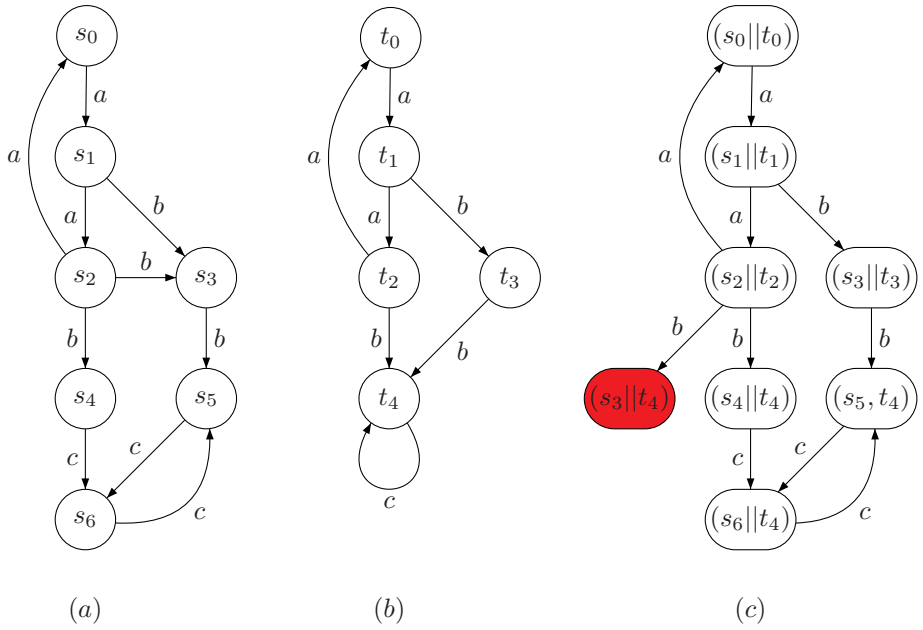

Fig. 1. An example DLTS (The red color stands for label 0 and white for label 1)

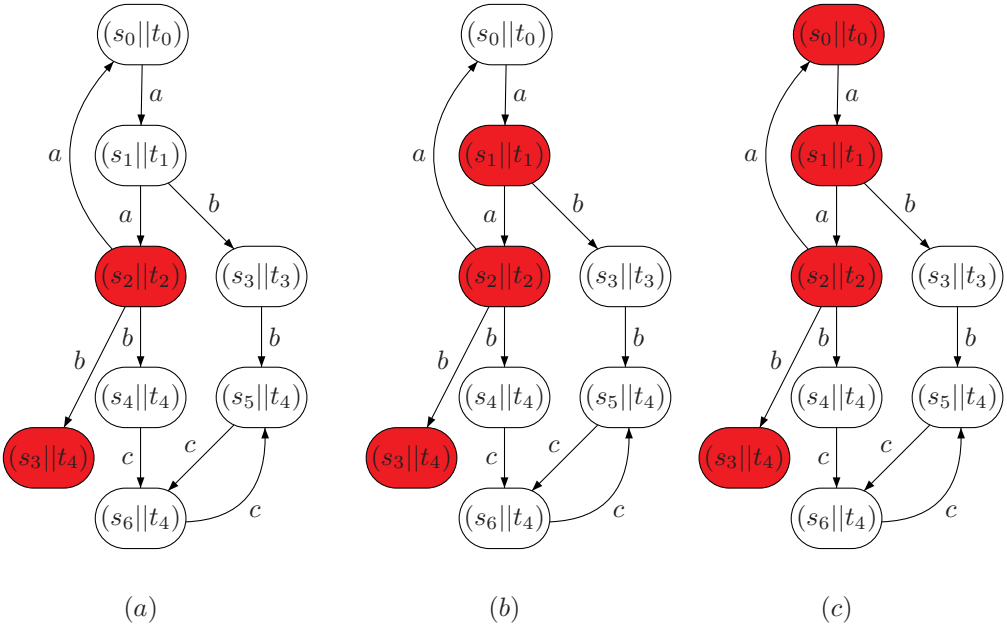

Fig. 2. Propagation of 0-labels

The number of candidate transitions is stored in $Ar_1[k,i,l]$. The initial value of $Ar_1[k,i,l]$, if it is not 0, is the number of times that the transition $s_k \xrightarrow{a}_1 s_i$ is used to derive transitions of the form $s_k||t_l \xrightarrow{a} s_i||t_j$ for some $j \in 1..n_2$. Similarly for the initialization of Ar_2. The intuition is that if there is a transition $(s_k||t_l) \xrightarrow{a} (s_i||t_j)$ in $\mathcal{L}$ then there exists the transition $t_l \xrightarrow{a}_2 t_j$ in $\mathcal{L}_2$ which can be used by t_l to simulate $s_k \xrightarrow{a} s_i$. In the procedure of propagating 0-labels, if $(s_i||t_j)$ has label 0, the value $Ar_1[k,i,l]$ decreases by 1 in the case of a simulation and $Ar_2[l,j,k]$ also decreases by 1 in the case of a bisimulation. The point is that if $s_i \not\sim t_j$ then $t_l \xrightarrow{a} t_j$ is not a suitable transition to mimic $s_k \xrightarrow{a} s_i$, thus the number of candidate transitions to mimic $s_k \xrightarrow{a} s_i$ decreases by 1. In the case of a bisimulation, we also have the dual phenomenon that $s_k \xrightarrow{a} s_i$ is not a suitable transition to mimic $t_l \xrightarrow{a} t_j$, which entails the decrement of $Ar_2[l,j,k]$. If $Ar_1[k,i,l]$ is equal to 0 after the decrement, we will relabel $(s_k||t_l)$ by 0 because the transition $s_k \xrightarrow{a} s_i$ cannot be matched up by any transition from t_l, i.e. $s_k \not\sim t_l$. In the case of a bisimulation, we will also relabel $(s_k||t_l)$ by 0 if $Ar_2[l,j,k] = 0$. Thus we have propagated the 0-label of the state $(s_i||t_j)$ to the state $(s_k||t_l)$.

In the algorithm we have chosen to use a breadth first traversal of $\mathcal{L}$ to look for 0-labelled states. An alternative way is to use a depth first traversal. As far as the consumption of time is concerned, there is no significant difference between the two options, at least for the examples we have tested (cf. Section V).

Proposition 3.9: Given two LTSs with initial states s_0 and t_0, the function **CheckBisim**(s_0, t_0) terminates, and it returns TRUE if and only if $s_0 \sim t_0$.

Proof: Termination is easy to be justified. Starting from some 0-labelled states, the algorithm tries to propagate 0-labels. Each time a state successfully propagates a 0-label to one of its predecessor, the total number of 1-labels in the DLTS decreases by 1. Eventually we must end up in one of the following two cases:

1) The state $(s_0||t_0)$ is reached and its label is successfully changed from 1 to 0. In this case **CheckBisim**(s_0, t_0) terminates and returns FALSE.
2) A stable situation is reached. That is, the procedure of propagation cannot progress because for any 0-labelled state $(s_i||t_j)$ with predecessor $(s_k||t_l)$ the values $Ar_1[k,i,l]$ and $Ar_2[l,j,k]$ do not diminish to 0. In this case the elements in stack St are kept popping out but no new element is pushed in. In the end the stack becomes empty and returns TRUE.

Algorithm 1 CheckBisim(s_0, t_0)

```
1: construct the DLTS L := L1||_R L2 and initialize Ar1 and Ar2.
2: if L(s0||t0)=0 then
3:    return FALSE
4: end if
5: initialize St := ∅ and L' := L
6: perform a breadth first traversal of L and push a pair (i,j) into St whenever L(si||tj) = 0
7: while St ≠ ∅ do
8:    (i,j) :=top(St); pop(St)
9:    for all (sk||tl) ∈ Pred(si||tj) with L'(sk||tl) = 1 do
10:      decrease both Ar1[k,i,l] and Ar2[l,j,k] by 1
11:      if Ar1[k,i,l] = 0 or Ar2[l,j,k] = 0 then
12:         set L'(sk||tl) := 0
13:         if (k,l) = (0,0) then
14:            return FALSE
15:         end if
16:         push (k,l) into St
17:      end if
18:   end for
19: end while
20: return TRUE
```

Therefore, the termination of our algorithm is guaranteed. It remains to prove the correctness. We show that **CheckBisim**(s_0, t_0) returns FALSE if and only if s_0 and t_0 are not bisimilar.

It is clear that **CheckBisim**(s_0, t_0) returns FALSE if and only if state $(s_0||t_0)$ has label 0. Nevertheless, we have the following property for the final DLTS when **CheckBisim**(s_0, t_0) terminates.

For any state $(s_k||t_l)$ in the final DLTS, it is given label 0 if and only if $s_k \not\sim t_l$.

The "only if" direction can be shown by induction as $(s_k||t_l)$ must have been reached by successive steps of label propagation.

1) The base case is that in the original DLTS state $(s_k||t_l)$ has label 0. Then by Definition 3.1 we have $Init(s_k) \neq Init(t_l)$. It immediately follows that $s_k \not\sim t_l$.
2) For the inductive step, suppose that a successor of $(s_k||t_l)$, say $(s_i||t_j)$, has changed the label of $(s_k||t_l)$ from 1 to 0. There must exist some $i \in 1..n_1$ or $j \in 1..n_2$ such that $Ar_1[k,i,l]$ or $Ar_2[l,j,k]$ decreases to 0. Without loss of generality, we assume that $Ar_1[k,i,l]$ decreases to 0. So s_k has the outgoing transition $s_k \xrightarrow{a} s_i$

for some action a. Let $(s_i||t_{j_1}), ..., (s_i||t_{j_m})$ be all the successors of $(s_k||t_l)$ with the first component being s_i. Then all these successors have label 0 before $(s_k||t_j)$ changes its label. By induction hypothesis $s_i \not\sim t_{j_p}$ for all $p \in 1..m$. So the transition $s_k \xrightarrow{a} s_i$ cannot be matched up by any candidate transition $t_l \xrightarrow{a} t_{j_p}$ from t_l, for all $p \in 1..m$. It follows that $s_k \not\sim t_l$.

For the "if" direction, suppose that $s_k \not\sim t_l$. Then $s_k \not\sim_n t_l$ for some $n \geq 1$. We can also show by induction on n that $(s_k||t_l)$ will be given label 0 in the final DLTS.

1) $n = 1$. Then $Init(s_k) \neq Init(t_l)$. In this case state $(s_k||t_l)$ has label 0 in the original DLTS. This label remains in the final DLTS since label propagation only changes a label from 1 to 0 but not the converse.
2) $n > 1$. Since $s_k \not\sim_n t_l$, there is a transition $s_k \xrightarrow{a} s_i$ that cannot be matched up by all the candidate transitions $t_l \xrightarrow{a} t_{j_1}$,...,$t_l \xrightarrow{a} t_{j_m}$ from t_l or the dual case that the candidate transitions from s_k cannot match up a transition from t_l. Without loss of generality we consider the first case. So $s_i \not\sim_{n-1} t_{j_p}$ for any $p \in 1..m$. By induction hypothesis at $(n-1)$, the state $(s_i||t_{j_p})$, which is a successor state of $(s_k||t_l)$, has label 0 for all $p \in 1..m$. Note that $Ar_1[k,i,l]$ has initial value j_m. When each successor state $(s_i||t_{j_p})$ tries to propagate its label, it decreases $Ar_1[k,i,l]$ by 1. Eventually we have $Ar_1[k,i,l] = 0$ and then the label of $(s_k||t_l)$ turns from 1 to 0.

■

Let us consider the time requirement of the algorithm. Let n_1 and m_1 (resp. n_2 and m_2) be the numbers of states and transitions in $\mathcal{L}_1$ (resp. $\mathcal{L}_2$), respectively. In addition, let n and m be the numbers of states and transitions in $\mathcal{L}$, respectively. Note that $n \leq n_1 n_2$ and $m \leq m_1 m_2$. Moreover, we have $m \geq n - 1$ since each state in $\mathcal{L}$ is reachable from (s_0, t_0). The time complexity for constructing the DLTS $\mathcal{L}$ is $O(m_1 m_2)$. The breadth first traversal of $\mathcal{L}$ takes time $O(n + m)$, which is equal to $O(m)$ as $m \geq n-1$. For the procedure of propagating 0-labels, a 0-labelled state enters the stack at most once, and when it is popped from the stack it checks all its predecessors. So the time requirement for the procedure of propagating 0-labels is bounded by $O(m)$. It follows that the worst case time complexity of the whole algorithm is $O(m_1 m_2)$.

Similar to the time requirement, the space requirement for constructing $\mathcal{L}$ is $O(m_1 m_2)$. The traversal of $\mathcal{L}$ takes space $O(m)$. The procedure of propagating 0-labels uses a stack with depth at most n and two arrays Ar_1, Ar_2 with sizes at most $n_1^2 n_2$ and $n_1 n_2^2$ respectively. The worst case space complexity of the whole algorithm is $O(m_1 m_2 + n_1^2 n_2 + n_1 n_2^2)$.

The algorithm proposed in [5] takes time $O(n^2)$ and space $O(n)$. Written in terms of n_i and m_i for $i = 1, 2$, they become $O(n_1^2 n_2^2)$ and $O(m_1 m_2)$. Therefore, in most cases where we have bounded fanout, i.e. there are at most a bounded number of transitions out of any state, we would have $m_i \leq c_i n_i$ for some constants c_i $(i = 1, 2)$ and the theoretical time complexity of our algorithm is better than that in [5] while our space complexity is worse.

If we omit Ar_2 from Algorithm 1 by deleting all underlined text, we obtain the algorithm for deciding simulations. Its correctness can be analysed analogously. Its worst case time and space complexities are $O(m_1 m_2)$ and $O(m_1 m_2 + n_1^2 n_2)$, respectively.

IV. Deterministic systems

If we restrict ourselves to deterministic LTSs, where performing an action from a state will lead to unique successor state, then the procedure **CheckBisi** in Algorithm 1 can be greatly simplified. There is no need of arrays Ar_1 and Ar_2 because if in the DLTS $\mathcal{L}$ a state $s_i||t_j$ has label 0 then we can directly propagate this label to all its predecessors. The reason is that the synchronous product of two deterministic LTSs is still deterministic, so if there is the transition $s_k||t_l \xrightarrow{a} s_i||t_j$ in $\mathcal{L}$ then $t_l \xrightarrow{a} t_j$ is the only candidate transition of $\mathcal{L}_2$ to simulate the transition $s_k \xrightarrow{a} s_i$ in $\mathcal{L}_1$. Therefore, if $s_i \not\sim t_j$ then surely we have $s_k \not\sim t_l$. As a consequence, **CheckBisi** can be simplified to be the following procedure **CheckBisi'**.

Let us have a look at the time complexity of the simplified algorithm. Note that the time requirement for constructing the product of $\mathcal{L}_1$ and $\mathcal{L}_2$ is now $O(\min(m_1, m_2))$ because a transition in $\mathcal{L}_1$ and $\mathcal{L}_2$ will be used at most once in forming $\mathcal{L}$. The traversal of $\mathcal{L}$ takes time $O(m)$ with $m \leq \min(m_1, m_2)$. The procedure of propagating 0-labels is bounded by $O(m)$. Hence, the worst case time complexity of the whole algorithm is $O(\min(m_1, m_2))$. By the way, since bisimilarity coincides with trace equivalence on deterministic LTSs, the above algorithm allows us to verify trace equivalence between deterministic LTSs in time $O(\min(m_1, m_2))$.

The space requirement for constructing $\mathcal{L}$ is $O(m)$. The traversal of $\mathcal{L}$ takes space $O(m)$. The stack St has depth at most n. Since $n - 1 \leq m \leq \min(m_1, m_2)$. It follows that the worst case space complexity of the algorithm is $O(\min(m_1, m_2))$.

The algorithm can also be used to check similarity, except that the construction of $\mathcal{L}$ is a bit different as

Algorithm 2 CheckBisim'(s_0, t_0)

1: construct the DLTS $\mathcal{L} := \mathcal{L}_1 || _R \mathcal{L}_2$.
2: **if** $L(s_0||t_0)$=0 **then**
3: **return** FALSE
4: **end if**
5: initialize $St := \emptyset$ and $L' := L$
6: perform a breadth first traversal of $\mathcal{L}$ and push a pair (i, j) into St whenever $L(s_i||t_j) = 0$
7: **while** $St \neq \emptyset$ **do**
8: (i, j) :=top(St); pop(St)
9: **for all** $(s_k||t_l) \in Pred(s_i||t_j)$ with $L'(s_k||t_l) = 1$ **do**
10: set $L'(s_k||t_l) := 0$
11: **if** $(k, l) = (0, 0)$ **then**
12: **return** FALSE
13: **end if**
14: push (k, l) into St
15: **end for**
16: **end while**
17: **return** TRUE

said in Definition 3.1. The worst case time and space complexities are still $O(\min(m_1, m_2))$.

Therefore, for verifying bisimilarity on deterministic LTSs, our algorithm is faster than Paige and Tarjan's partition refinement algorithm which takes time $O((m_1 + m_2) \log(n_1 + n_2))$. Moreover, the former can check similarity while the latter cannot.

V. Concluding remarks

We have presented a quasi-local algorithm that checks if two LTSs are related by bisimilarity or similarity with the worst case time complexity $O(m_1 m_2)$, where m_1 and m_2 are the numbers of transitions of the two LTSs. In the particular case of verifying bisimilarity or similarity on deterministic LTSs, the algorithm can be simplified and only takes time $O(\min(m_1, m_2))$. We have implemented our algorithm and tested it in a few simple examples like the jobshop and the alternating-bit protocol [12] where good performance of the algorithm is observed. In the example of alternating-bit protocol we show that an implementation of the protocol is weak bisimilar to its specification. In order to use our algorithm for checking weak bisimilarity, the original LTSs are saturated with weak transitions; computing the weak transition $(\xrightarrow{\tau})^*$ is achieved by a transitive closure algorithm (e.g. [1]). As to the future work, it would be very interesting to assess its performance on some practical examples with larger state spaces.

Acknowledgment

We thank Prof. Rudolf Fleischer for pointing out the related work on minimising DFA. We acknowledge the support of the National Natural Science Foundation of China (Grant No. 61033002) and the Opening Fund of Top Key Discipline of Computer Software and Theory in Zhejiang Provincial Colleges at Zhejiang Normal University.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.
[2] U. Celikkan. *Semantic Preorders in the Automated Verification of Concurrent Systems*. PhD thesis, North Carolina State University, 1995.
[3] R. Cleaveland and O. Sokolsky. *Equivalence and Preorder Checking for Finite-State Systems*, chapter 12, pages 391–424. North-Holland, 2001.
[4] A. Dovier, C. Piazza, and A. Policriti. An efficient algorithm for computing bisimulation equivalence. *Theoretical Computer Science*, 311(1-3):221–256, 2004.
[5] J.-C. Fernandez and L. Mounier. Verifying bisimulations "on the fly". In *Proceedings of the 3rd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 95–110. North-Holland, 1990.
[6] K. Fisler and M. Y. Vardi. Bisimulation minimization and symbolic model checking. *Formal Methods in System Design*, 21(1):39–78, 2002.
[7] R. Fleischer. State minimization. Lecture notes. Available at `http://www.tcs.fudan.edu.cn/rudolf/Courses/Theory/Theory09_cs/Resources/mini.pdf`.
[8] E. Kinber and C. Smith. *Theory of Computing: A Gentle Introduction*. Prentice Hall, 2000.
[9] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
[10] H. Lin. "On-the-fly instantiation" of value-passing processes. In *Proceedings of the International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, volume 135 of *IFIP Conference Proceedings*, pages 215–230. Kluwer, 1998.
[11] R. Mateescu and E. Oudot. Improved on-the-fly equivalence checking using boolean equation systems. In *Proceedings of the 15th International SPIN Workshop on Model Checking of Software*, volume 5156 of *Lecture Notes in Computer Science*, pages 196–213. Springer, 2008.
[12] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
[13] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.
[14] D. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.
[15] S. K. Shukla, H. B. H. III, and D. J. Rosenkrantz. HORNSAT, model checking, verification and games (extended abstract). In *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 99–110. Springer, 1996.
[16] B. Steffen and A. Ingólfsdóttir. Characteristic formulae for processes with divergence. *Information and Computation*, 110:149–163, 1994.