CHAPTER 4

VERIFICATION AND DIAGNOSIS OF TESTING PREORDER

4.1. Introduction

In protocol engineering, a common approach to system design and implementation is to verify if an implementation specification satisfies a given service specification. Here, the implementation specification represents a protocol specification or any lower level specification of a protocol, specifying how the functions should be implemented; while a service specification describes abstractly what functions a protocol should have. If an implementation specification does not satisfy its service specification, it is necessary to find out the faults and correct them. To verify that an implementation of a protocol satisfies its service specification, one needs to prove a conformance relation between the implementation specification and the service specification. Many conformance relations have been proposed in the literature for this purpose, among which are the testing equivalence and the testing preorder [Brin87], [Clea93].

A number of methods have been proposed in recent years to solve this problem. An algorithm is reported in [Clea93] for verifying if two specifications are testing equivalent: first the two specifications are transformed into acceptance graphs, which are deterministic and trace equivalent to their original specifications, with every state assigned an acceptance set; then an algorithm for the strong bisimulation relation [Miln89] is adapted for the
verification of testing equivalence. This algorithm is capable of verifying if two specifications satisfy the testing equivalence. However, when they are not related by the testing equivalence, the algorithm is unable to give diagnosis information indicating where the problem is. An algorithm is presented in [Celi92] for diagnosing the testing preorder. The algorithm first transforms the two specifications into acceptance graphs also, and then uses the algorithm proposed in [Celi91] to verify if the two acceptance graphs satisfy a modified bisimulation preorder. If they do not satisfy the bisimulation preorder, a postprocessing step is used to generate diagnostic information based on the results saved into a stack during the verification step.

The work presented in this chapter is motivated by the following observations:

1) The algorithms presented in [Clea93] and [Celi92] are unnecessarily complex both in time and space because of using the modified bisimulation algorithm. Protocol design and implementation are often involved in repeated debugging and revising, i.e., it is necessary to repeatedly apply the diagnosis algorithm to the two specifications. Therefore, it is important to improve the algorithm so that it is able to generate complete fault information, and is efficient both in time and space.

2) The definition of the testing preorder and testing equivalence in [Clea93] and [Celi92], although similar to the testing equivalence and the testing preorder defined in [Brin87] and [Nico87], discriminates two specifications that have different divergent properties. This may be unnecessary, since in protocol engineering, a correct specification of reliable message transmission through an unreliable channel always has some divergence, while its service specification may not have it. In addition, to detect which state is divergent needs significant computation.

Our contribution in this chapter can be summarized as follows:

1) A new method and a corresponding algorithm for the verification of testing preorder are presented. The algorithm reduces computation significantly, both in space and time, compared with the method proposed in [Clea93] and [Celi92]. The simplification is mainly due to the fact that we only transform the service specification, not the implementation, into a refusal graph for computing the testing preorder.
2) The algorithm produces a diagnostic information graph which contains complete fault information if an implementation does not conform to its service specification.

This chapter is organized as follows. In Section 4.2, the concept of refusal graph will be introduced, and an algorithm for the construction of the corresponding refusal graph for a given FLTS is proposed. In Section 4.3, an algorithm will be developed for verifying the testing preorder and producing diagnostic information. An example will be given to show the application in Section 4.4. Most of this chapter was published in [Tao95c].

4.2. The Definition of Refusal Graph

In this section, we define a special deterministic finite state machine, called refusal graph, shortly Rgraph, for a nondeterministic FLTS, similar to the definition of an acceptance graph defined in [Clea93] and [Celi92].

The following definition is used when we define an Rgraph for a given FLTS.

**Definition 4.1 (After_set):** Given an FLTS \( M = (Q, \Sigma, \delta, q_0) \) and \( \Sigma_0 / \Sigma \), we define the after set of a state \( p \in Q \) for \( \Sigma_0 \) as \( A_{\Sigma_0}(p) = \{ p' | p = \delta_{\Sigma_0}(p) \} \).

The After_set \( A_{\Sigma_0}(p) \) intuitively describes all the reachable states from a state \( p \) by executing zero, one or more events \( e(\Sigma - \Sigma_0) \approx \{ \tau \} \).

**Definition 4.2 (refusal graph):** An Rgraph is a 5-tuple \( G_{\Sigma_0} = (Q_g, \Sigma_0, \delta_g, R_g, s_0) \), where

- \( Q_g \) is a finite set of states.
- \( \Sigma_0 \) is a set of events.
- \( \delta_g: Q_g \times \Sigma_0 \times Q_g \) is a transition relation.
- \( R_g: Q_g \times (p(\Sigma_0)) \) is a mapping from \( Q_g \) to a set of subsets of \( \Sigma_0 \). For a state \( si \in Q_g \), \( R_g(si) \) is called a set of refusal sets.
- \( s_0 \in Q_g \) is the initial state.

This definition is similar to the definition of an FLTS. However, there are two differences: first, from the definition of the transition relation \( \delta_g: Q_g \times \Sigma_0 \times Q_g \), an Rgraph is deterministic; second, there is a mapping relation \( R_g: Q_g \times (p(\Sigma_0)) \) by which each state \( si \)
Qg is associated with a set of subsets of Σo, i.e., Rg(si), which will be explained by the following definition.

**Definition 4.3 (correspondence between an FLTS and an Rgraph):** Given an FLTS $M = (Q, Σ, δ, q0)$ and an Rgraph $G_{Σo}(M) = (Qg, Σo, δg, Rg, s0)$, where $Σo \sqcap Σ$, we say that $G_{Σo}(M)$ is the corresponding Rgraph of $M$ iff:

1) $Qg = \{ si | si = \{ qQ | q0 = t > Σo q \} \cap Tr_{Σo}(M) \}.$
2) $s0 = A_{Σo}(q0)$.
3) $∀si, sjQg$ and $∀ε Σo$, we have $sj ε ∅ si$ iff $si = \bigcup_{p' \in ψ} A_{Σo}(p')$,
   where $ψ = \{ p' | si \}$.
4) $∀siQg$, $Rg(si) = \{ Ref_{Σo}(M, p) | p \in si \}$.

Obviously, given a different $Σo$, we have a different Rgraph. In this definition, a set of states in $M$ is considered as one state in $G_{Σo}(M)$. This is similar to the method given in [Lewi81] for transforming a nondeterministic finite state machine to a trace equivalent deterministic finite state machine, except that [Lewi81] ignores the refusal sets. By ignoring the set of refusal sets for each state in $G_{Σo}(M)$ we get a deterministic FLTS, denoted as $P_{Σo}(M) = (Qg, Σo, δg, s0)$. Definition 4.3 implies that the Rgraph of a given FLTS is unique up to an isomorphism because of the following two facts:

1) If there are two corresponding Rgraphs, $G_{Σo}(M)$ and $G_{Σo}′(M)$, for a given FLTS $M$, then $Tr_{Σo}(G_{Σo}(M)) = Tr_{Σo}(G_{Σo}′(M))$.
2) For any trace $t$, if $s0 = t > Σo si$ in $G_{Σo}(M)$, then there is a state $s0′$ in $G_{Σo}(M)$ such that $s0′ = t > Σo si′$ and $Rg(si) = Rg(si′)$.

Therefore, we have the following lemma.

**Lemma 4.1 (uniqueness of the Rgraph):** For any FLTS, its corresponding Rgraph is unique up to isomorphism.

The following algorithm is developed to construct a refusal graph for a given FLTS $M$ according to Definition 4.3. The algorithm works as follows. In Step 1, a sub-algorithm, the AlgorithmRef(M), is used to obtain $Ref_{Σo}(M, p)$ for each state $p$ of $M$. It first computes the acceptance set $Acc_{Σo}(M, p)$ of a state $p$, by simply adding every observable event that can be enabled from a state $p'$ reachable from $p$ through executing a number of internal events. And then, compute $Ref_{Σo}(M, p) = Σo - Acc_{Σo}(M, p)$. This sub-algorithm can be
implemented more efficiently. However, for the sake of presentation, we do not optimize it in this thesis.

In Step 2, the initial state of the Rgraph is constructed by setting \( s_0 = A^{\Sigma_0}(p_0) \), and a set of refusal sets is associated with \( s_0 \). \( s_0 \) is marked TP, representing that this state will be processed further to create its outgoing transitions. In Step 3, new states and transitions are created from a state \( s_i \) marked TP. Each new state \( s_j \) contains a set of refusal sets computed from the states of \( M \) contained in \( s_j \). The new states are marked TP, and the state \( s_i \) is marked PD, meaning that all of its outgoing transitions have been created. The procedure continues until no state is marked TP.

**Algorithm RGraph(M)**

Input: An FLTS \( M = (Q, \Sigma, \delta, p_0) \);
Output: The refusal graph \( G = (Q_g, \Sigma_0, \delta_g, R_g, s_0) \) of \( M \);

Var: \( s_i, s_j \) /*state of a refusal graph*/
Var: \( p, p' \) /*state of \( M \)*/
Var: \( si(e) \) /*a set of states in \( M \)*/

**Begin**

1) Call AlgorithmRef(M) to compute \( \text{Ref}_{\Sigma_0}(M, p) \) for each state \( p \in Q \).

2) Compute \( s_0 = A^{\Sigma_0}(p_0) \), and mark it TP ("To be Processed").

Let \( R_g(s_0) = \{ \text{Ref}_{\Sigma_0}(M, p)|p \in s_0\} \);

3) Do the following while there is a state \( s_i \) marked TP:
   a) For every \( e \in \Sigma_0 \) do the following:
      i) Compute \( si(e)=p' \in \Psi \), where \( \Psi = \{ p | \exists \psi \text{ and } p-e \notin p' \} \).
      ii) If \( si(e) \) is not empty then
         If there is no previously created \( s_j \) containing exactly all the states in \( si(e) \), do the following:
         - Create such an \( s_j \) containing exactly all the states in \( si(e) \);
         - mark \( s_j \) TP and compute \( R_g(s_j) = \{ \text{Ref}_{\Sigma_0}(M, p)|p \in s_j\} \).

Create a transition labelled \( e \) from \( s_i \) to \( s_j \).
b) Change the mark of si from TP to PD. /* PD = Processed */

End

**AlgorithmRef(M)**

*Input:* \( M = (Q, \Sigma, \delta, p_0); \)

*Output:* \( \text{Ref}_{\Sigma_0}(M, p) \) is assigned to each state \( p \) of \( M \).

**Begin**

For every state \( p \in Q \), let Mark\([p]\) = NULL. /*NULL means that \( p \) is not marked*/

For every state \( p \in Q \)

Compute \( \text{Ref}_{\Sigma_0}(M, p) = \Sigma_0 - \text{Acc}_{\Sigma_0}(M, p) \);

**End**

**Acceptanceset(p)** /*used to compute \( \text{Acc}_{\Sigma_0}(M, p) \)*/

*Var:* mark[] /*an array of state marks*/

*Var:* \( \text{Acc}_{\Sigma_0}(M, p) \) /*a set of events, initially it is empty*/

**Begin**

1) Let \( \text{mark}[p] = p \);

2) For every \( e \in \Sigma_0 \) if there is a state \( p' \) such that \( p \rightarrow e \emptyset p' \) then add \( e \) to \( \text{Acc}_{\Sigma_0}(M, p) \);

3) For every \( p' \) and \( e \in \Sigma_0 \) such that \( p \rightarrow e \emptyset p' \), if \( \text{mark}[p'] = p \) then \( \text{Acc}_{\Sigma_0}(M, p) = \text{Acc}_{\Sigma_0}(M, p) \Rightarrow \text{Acceptanceset}(p') \);

Unmark all states.

Return \( \text{Acc}_{\Sigma_0}(M, p) \).

**End**

**Lemma 4.2:** The \( R \) graph computed by AlgorithmRgraph(M) for a given FLTS \( M \) satisfies Definition 4.3.
Proof: From Step 2 and 3, the Conditions 1 to 3 of Definition 4.3 are true. From Step 1 and $\text{Rg}(s_j) = \{\text{Ref}\Sigma_0(M, p) | p \in s_j\}$ of Step 2 and 3, the Condition 4 of Definition 4.3 is true.

The algorithm AlgorithmRgraph(M) has an exponential worst case complexity, due to the Step 4 that involves projecting a nondeterministic FLTS to a deterministic machine.

**Example 4.1:** For the given FLTS $M$ specified by Fig.4.1(a), where $\Sigma_0 = \{c, d, b\}$, the obtained refusal graph is shown in Fig.4.1(b). In Fig.4.1(b), we have the shadowed boxes: $s_0 = A\Sigma_0(0)$, $s_1 = A\Sigma_0(3)$, $s_2 = A\Sigma_0(4)$, $s_3 = A\Sigma_0(6)$. The refusal sets are shown beside each state of the RGraph.

![Diagram](attachment:image.png)

**Fig.4.1** (a) The specification $M$, (b) The Rgraph of $M$. 

59
Definition 4.4 (# coupled product between an FLTS and a refusal graph): Given $M = (Q, \Sigma, \delta, p_0)$ and a refusal graph $G_{\Sigma_0}(M') = (Q_g, \Sigma_0, \delta_g, R_g, s_0)$ for an FLTS $M'$, the # coupled product of $M$ and $G_{\Sigma_0}(M')$, written $M#G_{\Sigma_0}(M')$, is defined as $M#P_{\Sigma_0}(M')$, where $P_{\Sigma_0}(M')$ is obtained from $G_{\Sigma_0}(M')$ by ignoring the refusal set of each state, i.e., $P_{\Sigma_0}(M') = (Q_g, \Sigma_s, \delta_g, s_0)$.

We use this definition because it may remind us that $G_{\Sigma_0}(M')$ is a refusal graph and the set of refusal sets can be used. Similarly, we can define $\text{Tr}_{\Sigma_0}(G_{\Sigma_0}(M')) = \text{Tr}_{\Sigma_0}(P_{\Sigma_0}(M'))$.

4.3. Verification and Diagnosis of the Testing preorder

The purpose of the algorithm proposed in this section is: given an implementation specification $M = (Q, \Sigma, \delta, p_0)$ and a service specification $S = (Q_s, \Sigma_s, \delta_s, q_0)$, to check 1) $M \Sigma S$; 2) if $M \Sigma S$ is not true, generating diagnostic information indicating which states violate the conditions defined by Definition 2.6.

We have defined the concept of a refusal graph. An important property of a refusal graph is that: for any trace $t$ of $G_{\Sigma_s}(S) = (Q_g, \Sigma_s, \delta_g, R_g, s_0)$, there is only one state $siGg$ such that $s0 = t >_{\Sigma_s} si$. Constructing the product $M#G_{\Sigma_s}(S)$, for any trace $t \text{Tr}_{\Sigma_s}(M) \leftrightarrow \text{Tr}_{\Sigma_s}(S)$ and every state $p \in Q$ after $t$ in $M$, there is only one state $(p, si)$ in $M#G_{\Sigma_s}(S)$ such that $(q0, s0) = t > (p, si)$. If $M \Sigma S$, then according to the definition of the testing preorder, there is at least one refusal set $Rf Rg(si)$ such that $\text{Ref}_{\Sigma_s}(M, p) \prod Rf$ since $si$ is the only state such that $s0 = t > si$ in $G_{\Sigma_s}(S)$. In addition, to satisfy $\text{Tr}_{\Sigma_s}(M) \prod \text{Tr}_{\Sigma_s}(S)$, for any $(p, si)$ in $M#G_{\Sigma_s}(S)$ and any event $e \in \Sigma_s$, $p - e \emptyset p'$ in $M$ implies that there must be a state $(p', si')$ such that $(p, si) - e \emptyset (p', si')$ in $M#G_{\Sigma_s}(S)$. Based on this discussion, we have the following theorem.

**Theorem 4.1:** Given $M = (Q, \Sigma, \delta, p_0)$ and $S = (Q_s, \Sigma_s, \delta_s, q_0)$, let $G_{\Sigma_s}(S) = (Q_g, \Sigma_s, \delta_g, R_g, s_0)$ be the corresponding RGraph of $S$. $M \Sigma S$ iff every state $(p, si)$ of $M#G_{\Sigma_s}(S)$ satisfies the following two conditions:

1) For any event $e \in \Sigma_s$, $p - e \emptyset p'$ in $M$ implies that there is a state $(p', si')$ such that $(p, si) - e \emptyset (p', si')$ in $M#G_{\Sigma_s}(S)$.

2) There is at least one refusal set $Rf Rg(si)$ such that $\text{Ref}_{\Sigma_s}(M, p) \prod Rf$.

**Proof:** $(>)$ From Condition 1, we have $\text{Tr}_{\Sigma_s}(M) \prod \text{Tr}_{\Sigma_s}(S)$. For any $p \in Q$ in $M$, there must be a trace $t \text{Tr}_{\Sigma_s}(M) \leftrightarrow \text{Tr}_{\Sigma_s}(S)$ such that $p0 = t > p$, and there must be a state $(p, si)$ in $M#G_{\Sigma_s}(S)$ such that $(p0, s0) = t > (p, si)$, and $\forall qsi$ we have $q0 = t > q$ in $S$.
from the definition of $G_{\Sigma}(S)$, the definition of the $\#$ product, and the fact that $Tr_{\Sigma_{s}}(M) \prod Tr_{\Sigma_{s}}(S)$. From Condition 2, there is at least one state $qsi$ such that $Ref_{\Sigma_{s}}(S, q) Rg(si)$ and $Ref_{\Sigma_{s}}(M, p) \prod Ref_{\Sigma_{s}}(S, q)$. Therefore, $M_{\Sigma_{s}} S$ is proved.

(<) If $M_{\Sigma_{s}} S$, then for any $t$ $Tr_{\Sigma_{s}}(M) \leftrightarrow Tr_{\Sigma_{s}}(S)$ and any state $p$ after $t$ in $M$, there is a state $q$ after $t$ in $S$ such that $Ref_{\Sigma_{s}}(M, p) \prod Ref_{\Sigma_{s}}(S, q)$ according to Definition 2.6, and there must be a state $(p, si)$ in $M#G_{\Sigma_{s}}(S)$ such that $(p, s0) = t > (p, si)$ and $qsi$ according to the $\#$ product. That is to say, Condition 2 of Theorem 4.1 is true according to the definitions of $G_{\Sigma_{s}}(S)$ and $M#G_{\Sigma_{s}}(S)$. On the other hand, If $M_{\Sigma_{s}} S$, then $Tr_{\Sigma_{s}}(M) \prod Tr_{\Sigma_{s}}(S)$. If Condition 1 of Theorem 4.1 is not true, then there is a trace $t$ $Tr_{\Sigma_{s}}(M) \leftrightarrow Tr_{\Sigma_{s}}(S)$ and $e_{\Sigma_{s}}$ such that $p0 = t > p - e \emptyset$ and $t e Tr_{\Sigma_{s}}(S)$ according to the $\#$ product and the fact that $G_{\Sigma_{s}}(S)$ is deterministic. This is a contradiction with $Tr_{\Sigma_{s}}(M) \prod Tr_{\Sigma_{s}}(S)$. Therefore, Condition 1 of Theorem 4.1 is true.

This theorem gives us the idea how to develop an algorithm for verifying the testing preorder and generating diagnostic information: we can first construct the product $M#G_{\Sigma_{s}}(S)$, and then check if there is any state violating the two conditions of Theorem 4.1 in $M#G_{\Sigma_{s}}(S)$.

The algorithm works as follows. In Step 1, it computes $Ref_{\Sigma_{s}}(M, p)$ for each state $p$ of $M$ by using AlgorithmRef, and compute $G_{\Sigma_{s}}(S)$ of $S$ by using AlgorithmRGraph.

In the second step, $M#G_{\Sigma_{s}}(S)$ is computed, at the same time, the following two conditions are verified:

1) For every state $(p, si)$ in $M#G_{\Sigma_{s}}(S)$ and every $e_{\Sigma_{s}}$ such that $p - e \emptyset$ in $M$, check if $si - e \emptyset$ in $G_{\Sigma_{s}}(S)$.

2) For every state $(p, si)$ of $M#G_{\Sigma_{s}}(S)$, check if there is at least one refusal set $Rf Rg(si)$ such that $Ref_{\Sigma_{s}}(M, p) \prod Rf$.

Since an RGraph is also an FLTS except for the refusal sets associated with each state, we simply ignore the refusal set when computing $M#G_{\Sigma_{s}}(S)$.

If the first condition is violated at a state $(p, si)$, then $(p, si)$ is marked BD0 (meaning bad state of type 0); if the second condition is violated, then $(p, si)$ is marked BD1 (meaning bad state of type 1). We call a state a fault state if it is marked either BD0 or BD1. Each state ($p,$
si) contains a set of pointers pointing to its parent states (this will be used in Step 3). A variable FS is used to hold a set of pointers that point to the fault states, which will also be used in Step 3. If no state is marked BD0 or BD1, then \( M_{\Sigma S} \). Otherwise, the markings will be used in Step 3 to generate a diagnostic information graph by DiaInfo(M', FS), which removes all the states and related transitions in M' that can not reach any fault state without visiting the initial state. Hence, the final result contains all the traces from (p0, s0) to the fault states.

AlgorithmDiaTP(M, S)

\textbf{Input:} Protocol specification \( M = (Q, \Sigma, \delta, p0) \) and service specification \( S = (Q_s, \Sigma_s, \delta_s, q0) \).

\textbf{Output:} report \( M_{\Sigma S} \), or a diagnosis information graph.

\textbf{Var:} \( FS \) a set of pointers to states; /*each of the pointer points to a state*/
- \( p, q \) states in M and S;
- \( si \) a state in \( G_{\Sigma S}(S) \);
- \( (p, si) \) a state of \( M \#G_{\Sigma S}(S) \);
- \( e \) an event;
- \( Rf \) a set of events;

\textbf{Begin}

1) Computing \( Rf_{\Sigma S}(M, p) \) for every state \( p \) in M by AlgorithmRef, and compute the \( R \text{Graph} G_{\Sigma S}(S) = (Q_g, \Sigma_s, \delta_g, Rg, s0) \) of S by AlgorithmRGraph.

2) Let \( FS = \phi \), computing \( M' = M \#G_{\Sigma S}(S) \): the following two conditions are checked for each state \( (p, si) \) of \( M' \) during the computation:
   a) if \( p \rightarrow e \emptyset \) in M but \( si \rightarrow e \rightarrow \) in \( G_{\Sigma S}(S) \) for \( e \in \Sigma_s \), then mark \( (p, si) \) BD0 and create a pointer in FS which points to \( (p, si) \); otherwise
   b) if there is not a refusal set \( Rf \) \( Rg(si) \) such that \( Rf_{\Sigma S}(M, p) \prod Rf \), then mark state \( (p, si) \) BD1 and create a pointer in FS which points to \( (p, si) \).

3) If \( FS = \phi \), then report \( M_{\Sigma S} \); otherwise compute DiaInfo(M', FS).

\textbf{End}
DiaInfo(M', FS)

Begin

1) Mark (p0, s0) RF;
   /*An RF marks a state from which a fault state can be reached*/

2) While FS φ do the following
   a) Take a pointer from FS that points to a fault state (p, si) in M';
   b) Checkparent(p, si);

3) Remove from M' all the states that are not marked RF;

End

Checkparent(p, si)

Begin

Mark (p, si) RF;
For each parent state (p', si') of (p, si):
   if (p', si') is not marked RF then Checkparent(p', si')
Return

End

Theorem 4.2: MΣ S iff FS = φ at the end of Step 2 of AlgorithmDiaTP.

Proof: >) If FS = φ at the end of Step 2 of AlgorithmDiaTP, then TrΣs(M) ∏ TrΣs(S) from Step 2(a). Since TrΣs(M) ∏ TrΣs(S), for every t TrΣs(M) ↔TrΣs(S), there must be a state (p, si) such that (p0, s0) = t >(p, si) in M#GΣs(S), p0 = t >p in M and q0 = t >q in S for q si. If no state is marked BD1, then for every state (p, si) in M#GΣs(S), there is at least one state q si such that RefΣs(S, q)Rg(si) and RefΣs(M, p) ∏ RefΣs(S, q) according to Step 2(b). Therefore, MΣ S.

<) If MΣ S, then TrΣs(M) ∏ TrΣs(S). Hence, no state will be marked BD0 by Step 2(a) of the algorithm. For every state (p, si) in M#GΣs(S), there is a trace t TrΣs(M) ↔TrΣs(S) such that (p0, s0) = t >(p, si), p0 = t >p in M and q0 = t >q in S for q si from the definition of the # product and the definition of GΣs(S). According to the definition of the testing preorder, there is at least a state q si
such that $\text{Ref}_{\Sigma_s}(S, q) \cap \text{Rg}(si)$ and $\text{Ref}_{\Sigma_s}(M, p) \prod \text{Ref}_{\Sigma_s}(S, q)$. Therefore, no state will be marked BD1. That is, $FS = \emptyset$.

**Proposition 4.1:** The time and space complexity of AlgorithmDiaTP($M, S$) is $O(|Q|^{\infty}|Q_g|^{\infty}|Q_s|)$ in the worst case.

This can be proved by the fact that the main computation of the algorithm is in Step 2. The number of states of $M\#G_{\Sigma_s}(S)$ is at most $|Q|^{\infty}|Q_g|$. For any state $(p, si)$ of $M\#G_{\Sigma_s}(S)$, $si$ contains at most $|Q_s|$ refusal sets. Therefore, the computation needed in Step 2 is $O(|Q|^{\infty}|Q_g|^{\infty}|Q_s|)$ in the worst case.

In the worst case $|Q_g|$ has $2|Q_s|$ states. However, if $Q_s$ is small (this is true in protocol engineering since a service specification is much smaller than its corresponding protocol specification), the number of states $|Q_g|$ of $G_{\Sigma_s}(S)$ will not be large.

### 4.4. An Example

Given a system specification $M$ and a service specification $S$ which is supposed to be satisfied by $M$, as shown in Fig.4.2(a) and (b), respectively, we need 1) to verify whether $M_{\Sigma_s}S$; 2) if $M_{\Sigma_s}S$ is not true, then generate the diagnosis information. In Fig.4.2, the state 1 pointed by an arrow represents the initial state.

Let $\Sigma_s = \{a, b, c, e\}$, we obtain the Rgraph $G_{\Sigma_s}(S)$ as shown in Fig.4.3(a), in which the characters in $\{}$ beside a state $si$ represent the set of refusal sets $Rg(si)$. Fig.4.3(b) shows the results of Step 2, in which each state is named by two digits to represent a state $(p, si)$ of $M\#G_{\Sigma_s}(S)$. AlgorithmDiaTP finds two fault states: 95 and 75. For state 95, there is no refusal set $Rf$ in $Rg(5)$ such that $\text{Ref}_{\Sigma_s}(M, 9) \prod Rf$. Therefore, state 95 is marked BD1. For state 75, event $e$ can be enabled at state 7 in $M$, but no such an event can be enabled at state 5 of $G_{\Sigma_s}(S)$. Therefore, state 75 is marked BD0. Fig.3(b) is also the result of Step 3 in this example.

We modify the specification $M$ as shown in Fig.4.4(a). Applying the algorithm to Fig.4.4(a) and Fig.4.2(b), we have the result depicted in Fig.4.4(b). It is clear that the modified implementation specification and the service specification $S$ satisfy the testing preorder relation.
Fig. 4.2 (a) The implementation M; (b) The service S.
Fig. 4.3 (a) The Rgraph of $S$; (b) The output of the algorithm.
4.5. Discussion and Conclusions

In this chapter we have presented an efficient algorithm used for generating diagnostic information if two specifications do not satisfy the testing preorder [Brin87]. The basic method of verifying the testing preorder is to first transform the service specification $S$ into
its refusal graph, then to check and record any violation of the two conditions given in Theorem 4.1 for each state of the coupled product of the refusal graph and the implementation specification M. The advantages of our method are:

1) The time and space complexity of AlgorithmDiaTP is $O(|Q|\cdot|Qg|\cdot|Qs|)$. For the related work proposed in [Celi92], the time and space complexity is $O(|Q|\cdot|Qg|\cdot|Qg|\cdot|Qs|)$, where $|Qg|$ is the number of states of $G_{\Sigma}(M)$ (note that in [Celi92] it is claimed that the computation needed is $|Qg|\cdot|Qg|\cdot|Qg|$, this is not correct since during the computation of bisimulation of two acceptance graphs, each state of the two acceptance graphs contains a set of acceptance sets in dimension $|Q|$ and $|Qs|$ in the worst case, respectively. In addition, the computation of the postprocessing step is not counted there). Therefore, compared with the time and space complexity of the method proposed in [Celi92], AlgorithmDiaTP may save a factor of $O(|Qg|)$ in time and space in the worst case. Since $|Qg| = 2|Q|$ in the worst case, the improvement of our algorithm is significant. In addition, our algorithm does not need a complex postprocessing subalgorithm.

2) AlgorithmDiaTP is able to generate all fault information. However, the algorithm proposed in [Clea93] is not capable of producing the diagnosis information.