Computing Diagnostic Tests for Incorrect Processes*

Ufuk Celikkan, Rance Cleaveland
Department of Computer Science
N.C. State University
Raleigh, NC 27695-8206
{celikkan,rance}@science.csc.ncsu.edu

Abstract

This paper presents a method for generating diagnostic tests that is to be used in conjunction with an algorithm for computing the testing preorder of Hennessy and DeNicola. These tests may be used to debug a finite-state system that fails to meet (i.e be larger in the testing preorder than) a specification; in this case, the system is guaranteed to fail the test that is produced, while the specification is guaranteed to pass it. Our technique relies on transforming the diagnostic information produced by a general preorder algorithm into tests when the general preorder algorithm is used to compute the testing preorder. The method does not affect the time complexity of computing the relation.

1 Introduction

Researchers in process algebra have generally focused on the use of *behavioral relations* as a basis for specifying and verifying *reactive* systems such as communication protocols. One typical approach involves the use of a *preorder* to relate specifications (formulated as "underspecified" processes) and implementations (given as "fully defined" processes); a system is deemed correct if it is larger than its specification in the preorder, in which case it intuitively provides "at least" the behavior dictated by the specification. The testing preorders [7, 8] are particular examples of such relations; they are so named because they relate processes solely in terms of their responses to tests. As such, they have great intuitive appeal; if one process is larger than another in a testing preorder, then it is guaranteed to "pass" (in a precisely defined sense) all the tests than the lower process does. The testing preorders also have alternative characterizations that enable them to be computed for finite-state processes. At least two automated tools—the Concurrency Workbench [4] and Squiggles [1]— include algorithms for computing these relations.

Existing algorithms for the testing preorders are capable of indicating whether an implementation passes all the tests that its specification does. However, when this fails to be the case (i.e. when the specification can pass a test that the implementation can not), these algorithms are not capable of producing such a test: they can only signal that one exists. In general, exhibiting such a test can greatly ease the task of the system designer trying to debug the system, since by running the system on the test he or she can easily locate the source of error. Our goal in this paper, then, is to alter an existing algorithm for computing the testing preorder so that when one process fails

*Research supported by NSF/DARPA research grant CCR-901477, NSF grant CCR-9120995, ONR Young Investigator Award N00014-92-J-1582, and NSF Young Investigator Award CCR-9257963.

to be larger than another, such a diagnostic test is produced. Our technique relies on an algorithm that generates diagnostic information for a very general preorder, the *prebisimulation preorder* [12], which may be used as a basis for calculating a number of other behavioral preorders [4, 6]. We show how this information may be efficiently translated into tests when the general algorithm is used to compute the testing preorder.

Traditional Testing versus Testing-Based Verification

At this point it maybe useful to compare more traditional testing-based methodologies for reactive systems with the verification paradigm we pursue in this paper. In both cases the goal is similar: given a specification, one wishes to determine whether a system behaves correctly with respect to it, and if does not to determine what needs correction in the system. The differences surface in the ways in which this "system certification" take place. In test-generation-based approaches one builds a collection of tests from the specification (often in conjunction with the system) that the system is then subjected to. This set is necessarily finite, and thus for systems that should exhibit ongoing behaviour it can be difficult to ensure that enough tests are generated to establish unequivocally that the system is error-free. On the other hand, if a system fails one of the tests, one can use the test to diagnose the source of the error.

In verification-based approaches, especially in the one based on the testing preorder that we consider in this paper, the methodology ensures that when a system indeed meets its specification, any test the specification passes will also be passed by the system. However, traditional algorithms for computing these preorders generate no diagnostic information when a system is found not to satisfy a specification. The goal of the work in this paper is to marry the diagnostic advantage of test generation with the level of "coverage" intrinsic to verification by showing how testing-preorder algorithms may be altered to generate diagnostic tests.

The remainder of the paper is structured as follows. The next section develops our process model and reviews the definitions of the testing preorders and the prebisimulation preorder. The section following then briefly recaps the procedure of [3] for generating diagnostic information for the prebisimulation preorder. Section 4 presents our algorithm for building tests on the basis of this diagnostic information, and Section 5 gives examples illustrating our technique. The final section contains our conclusions.

2 Processes, Preorders and Equivalences

In this section we make precise our notion of process and define the various behavioral preorders and equivalences that we examine in this paper.

Definition 2.1 *A process is a quadruple $\langle P, Act, \rightarrow, p_0 \rangle$, where*

1. *P is a set of (process) states,*
2. *Act is a set of actions containing a distinguished* silent *action, τ,*
3. *$\rightarrow \subseteq P \times Act \times P$ is the* transition relation, *and*
4. *$p_0 \in P$ is the* start state.

Intuitively, P represents the set of possible states the process may be in, Act contains the actions that the process may perform, and $\rightarrow$ describes the state transitions that may result when the process executes an action in a state. The action τ represents an *internal* computation step. We shall write $p \xrightarrow{a} p'$ in lieu of $\langle p, a, p' \rangle \in \rightarrow$, and we shall on occasion use $p \xrightarrow{a}$ to mean that there is a p' with $p \xrightarrow{a} p'$ and $p \stackrel{a}{\not\rightarrow}$ if no such p' exists. If $p \stackrel{\tau}{\not\rightarrow}$, we say p is *stable*. In what follows we shall assume that $|P| < \infty$, i.e. that the set of states is finite.

For notational simplicity we use the following constructs to represent processes. (The interested reader should note the intended similarity with Milner's CCS [10].) *nil* is the process consisting of one state, an action set containing only τ and an empty transition relation. Intuitively, *nil* represents a terminated process that is incapable of any computation. Given $\mathcal{P} = \langle P, Act, \rightarrow, p_0 \rangle$, $a\mathcal{P}$ represents a process that initially performs a and then behaves like $\mathcal{P}$; its formal definition is given by

$$a\mathcal{P} = \langle P \cup \{p'\}, Act \cup \{a\}, \rightarrow \cup \{\langle p', a, p_0 \rangle\}, p' \rangle$$

where $p' \notin P$. Finally, given $\mathcal{P} = \langle P, Act_P, \rightarrow_P, p_0 \rangle$ and $\mathcal{Q} = \langle Q, Act_Q, \rightarrow_Q, q_0 \rangle$, $\mathcal{P} + \mathcal{Q}$ represents the process that is capable of behaving like $\mathcal{P}$ or $\mathcal{Q}$; its formal definition is

$$\begin{array}{rcl} \mathcal{P} + \mathcal{Q} & = & \langle \{ \langle 0, p \rangle \mid p \in P \} \cup \{ \langle 1, q \rangle \mid q \in Q \} \cup \{p'\}, \\ & & Act_P \cup Act_Q, \\ & & \{ \langle 0, p \rangle \xrightarrow{a} \langle 0, p' \rangle \mid p \xrightarrow{a}_P p' \} \cup \{ \langle 1, q \rangle \xrightarrow{a} \langle 1, q' \rangle \mid q \xrightarrow{a}_Q q' \} \cup \\ & & \quad \{ p' \xrightarrow{a} \langle 0, r \rangle \mid p_0 \xrightarrow{a}_P r \} \cup \{ p' \xrightarrow{a} \langle 1, r \rangle \mid q_0 \xrightarrow{a}_Q r \}, \\ & & p' \rangle. \end{array}$$

Notice that the states of $\mathcal{P}$ and $\mathcal{Q}$ are "tagged" with 0 and 1, respectively, in order to ensure that states with the same name in the two processes are differentiated. As an example, $a(bnil + cnil)$ is a process that can initially perform an a and then offers a choice of b or c. Frequently, when no confusion arises, we omit *nil*s and write e.g. $a(b + c)$.

Often we wish to abstract away from the internal (i.e τ) computations of a process $\mathcal{P}$ when analyzing its behavior. The observable transition relation $\Rightarrow$ is designed to support this. Let $s \in (Act - \{\tau\})^*$ be a sequence of visible actions, with p_1, p_2 states of $\mathcal{P}$. Then $p_1 \stackrel{s}{\Rightarrow} p_2$ is defined as follows.

- $p_1 \stackrel{\epsilon}{\Rightarrow} p_2$ if for some $n \geq 0$, $p_1 (\xrightarrow{\tau})^n p_2$. Here ϵ is the empty string and $(\xrightarrow{\tau})^n$ represents the n-fold composition of $\xrightarrow{\tau}$ with itself. Intuitively $p_1 (\xrightarrow{\tau})^n p_2$ holds if, starting in p_1, the system can perform n $\tau' s$ and evolve to p_2.

- $p_1 \stackrel{as}{\Rightarrow} p_2$ if there are p_1', p_2' such that $p_1 \stackrel{\epsilon}{\Rightarrow} p_1' \xrightarrow{a} p_2' \stackrel{s}{\Rightarrow} p_2$.

Intuitively, $p_1 \stackrel{s}{\Rightarrow} p_2$ holds if from p_1, the process can reach p_2 by executing the actions in s, with arbitrary amounts of internal computation allowed before and after each action.

Networks of processes may be constructed using various parallel composition operators [2, 10]. Generally, the semantics of these operators are defined in such a way that networks of concurrent processes may be reduced to single (albeit large) processes whose states are the global states of the networks. Accordingly, in this paper, we do not concern ourselves explicitly with concurrency.

2.1 Testing Preorders and Equivalences

Testing preorders and *equivalences* relate processes on the basis of their behavior in response to tests. In order to define particular testing preorders and equivalences, therefore, one must specify the *tests* of interest and a means for calculating a process' behavior *in response to* a test. One natural approach is taken by DeNicola and Hennessy [7, 8]: a test may be *any* process in which the set of actions Act contains a distinguished action w, the *success* action. Then, a process' behavior in response to a test is given in terms of the *computations* that result when the test is "applied" to the process. These intuitions may be formalized as follows.

Definition 2.2 *Let $\mathcal{P} = \langle P, Act, \rightarrow, p_0 \rangle$ be a process and $\mathcal{T} = \langle T, Act \cup \{w\}, \rightarrow, t_0 \rangle$ a test.*

1. *A* configuration *has the form $p||t$, where $p \in P$ and $t \in T$.*
2. *Configuration $p||t$ is* successful *if $t \xrightarrow{w}$.*
3. *The* configuration transition relation, *$p||t \xrightarrow{a} p'||t'$, is defined by the following rules.*
   - $a \neq \tau,\ p \xrightarrow{a} p', t \xrightarrow{a} t' \Rightarrow p||t \xrightarrow{a} p'||t'$.
   - $p \xrightarrow{\tau} p' \Rightarrow p||t \xrightarrow{\tau} p'||t$.
   - $t \xrightarrow{\tau} t' \Rightarrow p||t \xrightarrow{\tau} p||t'$.
4. *A* computation *of $p||t$ is a maximal (hence potentially infinite) sequence $p_1||t_1 \xrightarrow{a_1} p_2||t_2 \xrightarrow{a_2} \cdots$, where p_1 is p and t_1 is t. A computation is* successful *if for some $i \geq 1$, $p_i||t_i$ is successful.*
5. *p may t if $p||t$ has a successful computation. We also say that p may $\mathcal{T}$ if p may t_0, and $\mathcal{P}$ may $\mathcal{T}$ if p_0 may t_0.*
6. *p must t if every computation of $p||t$ is successful. In addition, p must $\mathcal{T}$ if p must t_0, and $\mathcal{P}$ must $\mathcal{T}$ if p_0 must t_0.*

Intuitively, the configuration transition relation describes the response of a process in state p to a test in state t. It is also designed to abstract away from the internal computation on the part of both the test and the process; either may evolve internally without interacting with the other. Also, since processes are in general nondeterministic, one can distinguish between the *possibility* of success ($\mathcal{P}$ *may* $\mathcal{T}$) and the *inevitability* of success ($\mathcal{P}$ *must* $\mathcal{T}$).

We now define the testing preorders and equivalences as follows. (Recall that a *preorder* is any reflexive and transitive relation.)

Definition 2.3 *Let $\mathcal{P}$ and $\mathcal{Q}$ be processes.*

1. *$\mathcal{P} <<_{may} \mathcal{Q}$ if for every test $\mathcal{T}$ such that $\mathcal{P}$ may $\mathcal{T}$, $\mathcal{Q}$ may $\mathcal{T}$.*
2. *$\mathcal{P} <<_{must} \mathcal{Q}$ if for every test $\mathcal{T}$ such that $\mathcal{P}$ must $\mathcal{T}$, $\mathcal{Q}$ must $\mathcal{T}$.*
3. *$\mathcal{P} <<_t \mathcal{Q}$ if $\mathcal{P} <<_{may} \mathcal{Q}$ and $\mathcal{P} <<_{must} \mathcal{Q}$.*
4. *Let $i \in \{may, must, test\}$. Then $\mathcal{P} =_i \mathcal{Q}$ if $\mathcal{P} <<_i \mathcal{Q}$ and $\mathcal{Q} <<_i \mathcal{P}$.*

So $\mathcal{P} <<_{may} \mathcal{Q}$ if $\mathcal{Q}$ may pass all the tests that $\mathcal{P}$ may pass, while $\mathcal{P} <<_{must} \mathcal{Q}$ if $\mathcal{Q}$ must pass all the tests that $\mathcal{P}$ must pass.

2.2 The Prebisimulation Preorder

As we indicated in the introduction, the testing preorders may be characterized in terms of the prebisimulation preorder. In this section we review the definition of this preorder; the interested reader is referred to [12] for details.

The prebisimulation preorder, $\sqsubset\!\!\!\sim$, relates processes in terms of relations called *prebisimulations*. Intuitively, a prebisimulation is a "matching" between states of processes $\mathcal{P}$ and $\mathcal{Q}$ that satisfies a couple of conditions. The first stipulates that if state p is matched to state q, then each a-transition of p must be matched by some a-transition of q. The second condition requires that each a-transition of q be matched by some a-transition of p, *provided that* the behavior of p is "completely defined" with respect to a. In other words, if the a-transitions of p are only "partially specified", then p is not required to match the a-transitions of q. Formally, let $\Uparrow \subseteq P \times Act$ be a predicate called the *divergence predicate*; we shall write $p{\Uparrow}a$ in lieu of $\langle p, a\rangle \in \Uparrow$ and $p{\Downarrow}a$ in lieu of $\neg(p{\Uparrow}a)$. ($\Downarrow$ is often called a *convergence predicate.*) Intuitively, if $p{\Uparrow}a$ is true then the behavior of p on a should be thought of as incompletely defined, and thus a prebisimulation relating p to q need not match every a-transition of q with some a-transition in p. Traditionally, divergence is associated with infinite internal computation [8, 12]. In CCS, for example, $p{\Uparrow}a$ holds if p can engage in an infinite sequence of τ's or if for some p' such that $p \stackrel{a}{\Rightarrow} p'$, p' can perform an infinite sequence of $\tau' s$.

Definition 2.4 *Let $\mathcal{P} = \langle P, Act, \rightarrow, p_0\rangle$ and $\mathcal{Q} = \langle Q, Act, \rightarrow, q_0\rangle$ be processes with $\Downarrow \subseteq (P \cup Q) \times Act$ a convergence relation. A relation $R \subseteq P \times Q$ is a prebisimulation between $\mathcal{P}$ and $\mathcal{Q}$ if pRq implies the following.*

1. $p \stackrel{a}{\rightarrow} p' \Rightarrow \exists q'.\ q \stackrel{a}{\rightarrow} q' \wedge p'Rq'$.
2. $p{\Downarrow}a \Rightarrow [q{\Downarrow}a \wedge (q \stackrel{a}{\rightarrow} q' \Rightarrow \exists p'.\ p \stackrel{a}{\rightarrow} p' \wedge p'Rq')]$.

We may now define $\sqsubset\!\!\!\sim$ as follows.

Definition 2.5 *Let $\mathcal{P} = \langle P, Act, \rightarrow, p_0\rangle$ and $\mathcal{Q} = \langle Q, Act, \rightarrow, q_0\rangle$ be processes. Then $\mathcal{P} \sqsubset\!\!\!\sim \mathcal{Q}$ if and only if there is a prebisimulation R between $\mathcal{P}$ and $\mathcal{Q}$ with $p_0 R q_0$.*

In this paper, we use a slight refinement of the usual notion of prebisimulation just given. Let $\mathcal{P} = \langle P, Act, \rightarrow, p_0\rangle$ and $\mathcal{Q} = \langle Q, Act, \rightarrow, q_0\rangle$ be processes, and let $\Pi \subseteq P \times Q$ be a relation. Intuitively, Π is supposed to be a "compatibility relation" between states of $\mathcal{P}$ and $\mathcal{Q}$. Sometimes, states contain information in addition to their outgoing transitions that is of interest, and Π determines when this extra information in two states is compatible. Then a relation R is a Π-prebisimulation if $R \subseteq \Pi$ and R is a prebisimulation. So if pRq, then p and q must be related by Π in addition to having their transitions matched appropriately by R. We write $\mathcal{P} \sqsubset\!\!\!\sim_{\Pi} \mathcal{Q}$ when there is a Π-prebisimulation relating p_0 and q_0. Note that when Π is itself a preorder, then so is $\sqsubset\!\!\!\sim_{\Pi}$.

The prebisimulation preorder is interesting in its own right, and efficient algorithms exist for computing it [5]. In addition, it may be used as the basis for deciding a number of other behavioral relations. Accordingly, an algorithm for computing it has been implemented in the Concurrency Workbench [4].

A Logical Characterization of $\lesssim_\Pi$

The prebisimulation preorder also has a characterization in terms of formulas of a particular logic; this formulation states that $\mathcal{P} \lesssim \mathcal{Q}$ holds if whenever $\mathcal{P}$ satisfies a formula, then $\mathcal{Q}$ does as well. The logic is a variant of the Hennessy-Milner logic [9]; the syntax of formulas may be given as follows.

$$\Phi ::= tt \mid ff \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \langle a \rangle \Phi \mid [a]_{\Downarrow}\Phi$$

The logic is a modal logic, and the formulas are meant to be interpreted with respect to states in a given process. Intuitively, tt and ff are true of every state and of no state respectively. A state satisfies $\Phi_1 \vee \Phi_2$ if it satisfies either Φ_1 or Φ_2, while it satisfies $\Phi_1 \wedge \Phi_2$ if it satisfies both. The formulas $\langle a \rangle \Phi$ and $[a]_{\Downarrow}\Phi$ may be used to make statements about a process' a-transitions. A state satisfies the former if it has an a-transition satisfying Φ, while it satisfies the latter if all its a-transitions satisfy Φ, *and the state is a-convergent.* We say that a process satisfies a formula if its start state does. Formally, the satisfaction relation, $\models$, relates states in a process $\mathcal{P} = \langle P, Act, \rightarrow, p_0 \rangle$ to formulas, and it is defined to be the smallest relation satisfying the following, where $p \in P$:

$p \models tt$
$p \models \Phi_1 \vee \Phi_2$ if $p \models \Phi_1$ or $p \models \Phi_2$
$p \models \Phi_1 \wedge \Phi_2$ if $p \models \Phi_1$ and $p \models \Phi_2$
$p \models \langle a \rangle \Phi$ if $\exists q \;\; p \xrightarrow{a} q$ and $q \models \Phi$
$p \models [a]_{\Downarrow}\Phi$ if $p{\Downarrow}a$ and $\forall q$ if $p \xrightarrow{a} q$, then $q \models \Phi$

We say that $\mathcal{P} \models \Phi$ if $p_0 \models \Phi$. Stirling [11] proves the following theorem.

Theorem 2.6 *Let $\mathcal{P}$ and $\mathcal{Q}$ be processes. Then $\mathcal{P} \lesssim \mathcal{Q}$ if and only if any formula Φ in the above logic satisfied by $\mathcal{P}$ is also satisfied by $\mathcal{Q}$.*

Thus, if $\mathcal{P} \not\lesssim \mathcal{Q}$, then there must be a formula that $\mathcal{P}$ satisfies and $\mathcal{Q}$ does not. This suggests that formulas may be used as diagnostic information; if it turns out that a system *Sys* fails to meet its specification *Spec* (i.e. $Spec \not\lesssim Sys$), then there is a formula that *Spec* satisfies and *Sys* does not.

In the case of $\lesssim_\Pi$, the logic must be extended slightly to get a theorem analogous to Theorem 2.6. The new formulas needed are those of the form $\Pi(p)$, where p is a state in P. Intuitively, a state r satisfies $\Pi(p)$ if $p \Pi r$. Sometimes, when $q \not\models \Pi(p)$, we annotate $\Pi(p)$ with q and write $\Pi_q(p)$. This does not change the meaning of the formula, but it can be convenient for the purposes of generating diagnostic information.

2.3 Testing via Prebisimulation

We now briefly review how the testing preorders may be characterized in terms of the prebisimulation preorder. This characterization is of interest because it permits prebisimulation algorithms to be used to compute the testing preorders. In this section, and in the remainder of the paper, we focus on the testing preorder, $<\!\!<_t$; the analysis of $<\!\!<_{may}$ and $<\!\!<_{must}$ is similar.

The characterization relies on the definition of a *process transformation*, $\mathcal{A}$, and relation Π having the property that $\mathcal{P} <\!\!<_t \mathcal{Q}$ if and only if $\mathcal{A}(\mathcal{P}) \lesssim_\Pi \mathcal{A}(\mathcal{Q})$. Following [6, 4] we refer to $\mathcal{A}(\mathcal{P})$ as the *acceptance graph* for $\mathcal{P}$.

Before giving the formal definition of $\mathcal{A}$ we will first concentrate on the intuition underlying the construction. There is an alternative characterization of the testing preorder [8] that relies on three characteristics of a process $\mathcal{P} = \langle P, Act, \rightarrow, p_0 \rangle$.

- The *language* of $\mathcal{P}$. The language of a process is the set of traces, or sequences of visible actions (i.e. τ actions are omitted), that are possible from the start state of a process. Formally, the language $L(\mathcal{P})$ of $\mathcal{P}$ is defined by $L(\mathcal{P}) = \{\, s \in (Act - \{\tau\})^* \mid \exists p'.\, p_0 \stackrel{s}{\Rightarrow} p' \,\}$.

- The *degree of nondeterminism* of $\mathcal{P}$. Because of potential nondeterminism, after "executing" a visible sequence of actions s, $\mathcal{P}$ may be one of several states, each of which may enable a different set of actions. This information may be captured using *acceptance sets*. Intuitively, the acceptance set $\mathcal{A}(\mathcal{P}, s)$ of $\mathcal{P}$ after a sequence s is a set of sets of actions; each component set contains the actions enabled in one stable state that $\mathcal{P}$ may be in after s. The formal definition of $\mathcal{A}(\mathcal{P}, s)$ is: $\mathcal{A}(\mathcal{P}, s) = \{\, init(p') \mid p_0 \stackrel{s}{\Rightarrow} p' \wedge p' \stackrel{\tau}{\not\rightarrow} \,\}$, where $init(p') = \{\, a \mid p' \stackrel{a}{\rightarrow} \,\}$.

- The *divergence potential* of $\mathcal{P}$. Some states of $\mathcal{P}$ may be capable of an infinite sequence of τ-transitions, and during this computation it may refuse any action offered by a test. Thus, while executing a sequence s, $\mathcal{P}$ may at some point be incapable of any further interaction because of this divergence. Formally, given $s \in (Act - \{\tau\})^*$, we define $\mathcal{P} \Uparrow s$ to hold if $p_0 \Uparrow s$, where $p \Uparrow s$ is given inductively by :

  - $p \Uparrow \epsilon$ if p can do an infinite τ-computation,
  - $p \Uparrow as$ if $p \Uparrow \epsilon$ or if there is a p' such that $p \stackrel{a}{\Rightarrow} p'$ and $p' \Uparrow s$.

The transformation $\mathcal{A}$ records this information as follows. First, it first "determinizes" $\mathcal{P}$ using a variant of the standard determinization procedure for finite-state machines (the variation comes in the way divergence is treated [6]). In the resulting transformed process, each state corresponds to a set of states from $\mathcal{P}$ that is "ϵ-closed" in the sense that if p is in a set, and $p \stackrel{\epsilon}{\Rightarrow} p'$, then p' is also in the set. The start state in the new process corresponds to the ϵ-closure of the set containing only the start state of $\mathcal{P}$. Then, each "determinized" state y is labeled with two pieces of information: an acceptance set recording the initial moves of each (stable) state in P that corresponds to y, and a boolean indicating whether any state associated to it perform an infinite sequence of τs.

The formal definition of $\mathcal{A}(\mathcal{P}) = \langle Y, Act, \rightarrow_Y, y_0 \rangle$ given below is taken from [6]; we use the following notation where $Q \subseteq P$ is a set of states in $\mathcal{P}$.

- $Q^\epsilon = \{\, p \mid \exists q \in Q.\, q \stackrel{\epsilon}{\Rightarrow} p \,\}$;
- $Q \Downarrow$ exactly when $\forall q \in Q \,.\, q \Downarrow$, and $Q \Uparrow$ if $\neg(Q \Downarrow)$;
- $D(Q, a) = \bigcup_{q \in Q} \{\, q' \mid q \stackrel{a}{\rightarrow} q' \,\}$.

Definition 2.7 *Let $\mathcal{P}$ be a process. The acceptance graph $\mathcal{A}(\mathcal{P}) = \langle Y, Act, \rightarrow_Y, y_0 \rangle$ of $\mathcal{P}$ is defined as follows.*

1. $Y = \{\, \langle Q, b \rangle \in 2^P \times boolean \mid Q = Q^\epsilon \wedge (b = true \Rightarrow Q \Downarrow) \,\}$.
2. *For $y = \langle Q, b \rangle \in Y$, define $y.conv = b$ and*

$$y.acc = \begin{cases} \emptyset & \text{if } y.conv = false \\ \{\, init(q) \mid q \in Q \wedge q \stackrel{\tau}{\not\rightarrow} \,\} & \text{otherwise} \end{cases}$$

3. *For $y_1 = \langle Q_1, b_1 \rangle$ and $y_2 = \langle Q_2, b_2 \rangle$, $y_1 \stackrel{a}{\rightarrow} y_2$ exactly when the following hold.*

(a) $a \neq \tau$.

(b) $(D(Q_1, a))^\epsilon = Q_2$

(c) $[b_1 = \mathit{false} \Rightarrow b_2 = \mathit{false}] \wedge [(b_1 = \mathit{true} \wedge b_2 = \mathit{false}) \Rightarrow Q_2 \Uparrow]$.

4. $y_0 = (\{p_0\})^\epsilon$.

Note that the acceptance set of an acceptance graph state also encodes its divergence information, since $y.acc = \emptyset$ iff $y.conv$ is false. We now define the relation Π that we use to determine when two states in an acceptance graph are "compatible". Π relates two states, y and z, on the basis of their acceptance sets and divergence information. Let $y.acc$ and $y.conv$ be the acceptance set and convergence boolean of y. Then $y \Pi z$ [1] holds if:

- $y.conv$ is false (i.e. y is divergent), or $z.conv$ is true; and
- if $y.conv$ is true, then each set in $z.acc$ has a subset in $y.acc$. In this case we write (following [8]) $z.acc \subset\subset y.acc$. $z.acc \subset\subset y.acc$ if for every $S \in z.acc$, there is an $S' \in y.acc$ with $S' \subseteq S$. Intuitively $z.acc \subset\subset y.acc$ holds if the states associated to z are "more deterministic" than those associated to y.

Also define $y \Uparrow a$ to hold if $y.conv$ is false. (Notice that for acceptance graphs, the "a" parameter is not used.) As an illustration of these definitions consider the process $\mathcal{P}$ and its acceptance graph $\mathcal{A}(\mathcal{P})$ from [6] in Figure 1. We have that

$L(\mathcal{P}) = (abc)^*\{\epsilon, a, aa, ab, ac, abc\}$
$\mathcal{A}(\mathcal{P}, a) = y_1.acc = \{\{b, c\}, \{a, c\}\}$
$y_1.acc \subset\subset y_2.acc$
$y_5.conv = false$
state y_1 in $\mathcal{A}(\mathcal{P})$ corresponds to $\{p_1, p_2, p_5\}$ in $\mathcal{P}$.

Cleaveland and Hennessy [6] prove the following.

Theorem 2.8 *Let $\mathcal{P}$ and $\mathcal{Q}$ be processes. Then $\mathcal{P} <<_t \mathcal{Q}$ if and only if $\mathcal{A}(\mathcal{P}) \sqsubseteq_\Pi \mathcal{A}(\mathcal{Q})$.*

3 Computing the Preorder and Generating Diagnostic Information

The preorder $\sqsubseteq_\Pi$ has an iterative characterization that makes it suitable for algorithmic computation [12]. The algorithm used in this paper is based on this characterization, which is as follows:

Definition 3.1 *Let $\Pi \subseteq P \times Q$. A family of relations $\sqsubseteq^i_\Pi$ over $P \cup Q$ can be defined as follows:*

1. $\sqsubseteq^0_\Pi = \Pi$.
2. $\sqsubseteq^{i+1}_\Pi = \mathcal{F}(\sqsubseteq^i_\Pi)$ *where* $\mathcal{F}(R) = \{\langle p, q\rangle \mid$

$$(p \xrightarrow{a} p' \Rightarrow \exists q'.\, q \xrightarrow{a} q' \wedge p' R q') \wedge$$

[1] It should be noted that this definition of Π differs slightly from the one given in [6]. However, one can easily show that this difference is inconsequential to the definition of $\sqsubseteq_\Pi$.

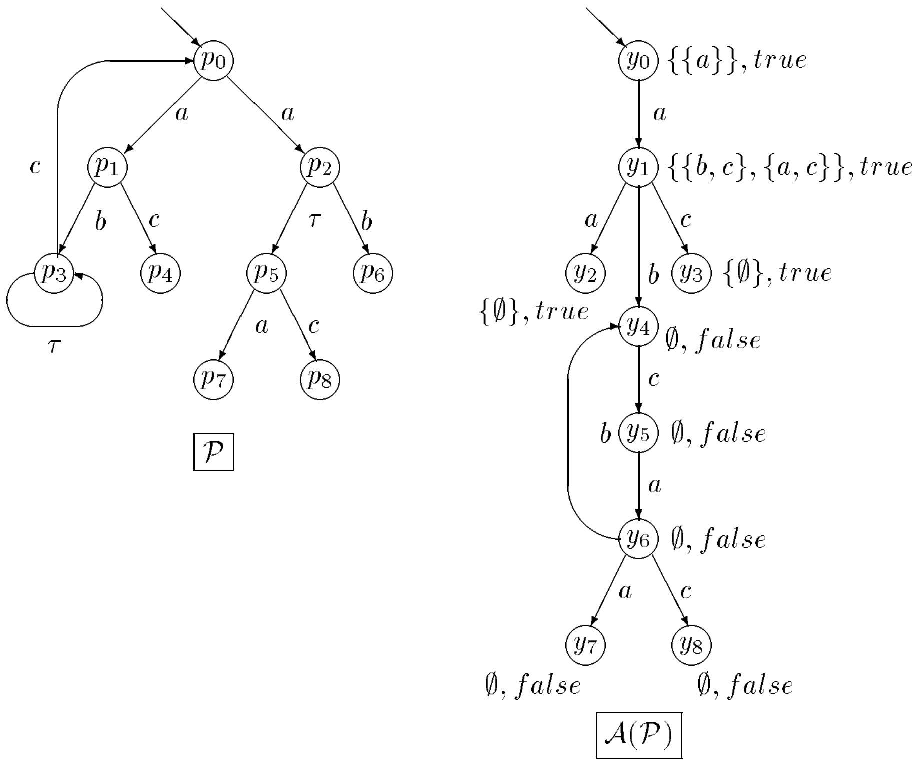

Figure 1: An example.

$$(p{\Downarrow}a \Rightarrow [q{\Downarrow}a \wedge (q \xrightarrow{a} q' \Rightarrow \exists p'.\, p \xrightarrow{a} p' \wedge p'Rq')])\}.$$

Then $p \underset{\sim}{\sqsubset}_{\Pi} q$ *iff* $\forall i\ p \underset{\sim}{\sqsubset}^{i}_{\Pi} q$.

Note that if R is a Π-prebisimulation then $R \subseteq \mathcal{F}(R)$, and that $p \not\underset{\sim}{\sqsubset}^{i}_{\Pi} q$ implies $p \not\underset{\sim}{\sqsubset}^{i+1}_{\Pi} q$. Thus $\underset{\sim}{\sqsubset}_{\Pi}$ can be computed by repeatedly applying $\mathcal{F}$ to Π until $\underset{\sim}{\sqsubset}^{i+1}_{\Pi} = \underset{\sim}{\sqsubset}^{i}_{\Pi}$. This suggests the algorithm *PREBISIM* in Figure 2 for constructing $\underset{\sim}{\sqsubset}_{\Pi}$.

Computing Distinguishing Formulas

From the definition of $\mathcal{F}$ it follows that if $p \not\underset{\sim}{\sqsubset}^{i}_{\Pi} q$ then one of the six conditions below must hold.

0. $p \not\Pi q$ $(i = 0)$

1. $p \xrightarrow{a} p'$ but $q \stackrel{a}{\not\rightarrow}$

2. $p \xrightarrow{a} p'$ and $q \xrightarrow{a}$ but $\forall q'$. $(q \xrightarrow{a} q'$ implies $p' \not\underset{\sim}{\sqsubset}^{i-1}_{\Pi} q')$

3. $p{\Downarrow}a$ but $q{\Uparrow}a$

4. $p{\Downarrow}a$, $q{\Downarrow}a$, and $q \xrightarrow{a} q'$, but $p \stackrel{a}{\not\rightarrow}$

$PREBISIM(\Pi; \mathcal{P} : \langle P, Act, \rightarrow, p_0\rangle; \mathcal{Q} : \langle Q, Act, \rightarrow, q_0\rangle)$;
  $\precsim_\Pi^0 := \Pi$;
  $\precsim_\Pi^1 := \mathcal{F}(\precsim_\Pi^0)$;
  $k := 1$;
  while $\precsim_\Pi^k \neq \precsim_\Pi^{k-1}$ **do**
    $\precsim_\Pi^{k+1} := \mathcal{F}(\precsim_\Pi^k)$;
    $k := k + 1$;
  end
  if $p_0 \precsim_\Pi^k q_0$ **then return** *true*
  else return *false* ;
end *PREBISIM* ;

Figure 2: Algorithm *PREBISIM*.

5. $p{\Downarrow}a$, $q{\Downarrow}a$, $q \xrightarrow{a} q'$, and $p \xrightarrow{a}$, but $\forall p'. (p \xrightarrow{a} p'$ implies $p' \not\precsim_\Pi^{i-1} q')$

Saving this information appropriately allows us to compute distinguishing formulas; i.e if $p \not\precsim_\Pi q$ then by using the saved information we may generate a formula Φ such that p satisfies Φ but q does not. Accordingly, the algorithm *PREBISIM* may be modified by adding a stack that holds tuples recording the information necessary to compute distinguishing formulas. As the algorithm computes $\precsim_\Pi^i$ it records state pairs $\langle p, q\rangle$ such that $p \not\precsim_\Pi^i q$. It does so by pushing a tuple containing $\langle p, q\rangle$, a tag 0-5 indicating which condition above ensures $p \not\precsim_\Pi^i q$, and additional information depending on the tag. The specific tuple formats are :

0) $[0, \langle p, q\rangle]$

1) $[1, \langle p, q\rangle, a, p']$

2) $[2, \langle p, q\rangle, a, p']$

3) $[3, \langle p, q\rangle, a]$

4) $[4, \langle p, q\rangle, a, q']$

5) $[5, \langle p, q\rangle, a, q']$

For example, if $p \xrightarrow{a} p'$ but $q \not\xrightarrow{a}$ then $[1, \langle p, q\rangle, a, p']$ will be pushed into the stack, and if $p{\Downarrow}a$ but $q{\Uparrow}a$ then $[3, \langle p, q\rangle, a]$ will be pushed into stack.

PREORDER in Figure 3 is the algorithm *PREBISIM* altered to maintain the stack. Given this stack computed by *PREORDER*, if two start states p_0 and q_0 are not related by $\precsim_\Pi$, then there is a tuple containing p_0 and q_0 in the stack, and the algorithm DFG (pseudo-code for which is contained in Appendix A) will build a formula satisfied by p_0 but not q_0. The main idea may be illustrated using the following example. Let $[2, \langle p, q\rangle, a, p']$ be in the stack. From the way this item is generated we know that p has an $a-$derivative to p' and it is not related to any one of the $a-$derivatives of q. Accordingly if one recursively builds formulas that distinguish p' from each $a-$derivative of q and takes their conjunction (call it Φ) then p satisfies $\langle a\rangle\Phi$ and q does not.

$PREORDER(\Pi; \mathcal{P} : \langle P, Act, \rightarrow, p_0\rangle; \mathcal{Q} : \langle Q, Act, \rightarrow, q_0\rangle)$;
  $\sqsubseteq^0_\Pi := \Pi$;
  for all p and q such that $p \not\sqsubseteq^0_\Pi q$
    $stack := PUSH([0, \langle p, q\rangle], stack)$;
  $\sqsubseteq^1_\Pi := \mathcal{F}(\sqsubseteq^0_\Pi)$;
  $k := 1$;
  while $\sqsubseteq^k_\Pi \neq \sqsubseteq^{k-1}_\Pi$ **do**
    foreach $\langle p, q\rangle$ such that $p \sqsubseteq^{k-1}_\Pi q$ but $p \not\sqsubseteq^k_\Pi q$ **do**
      case
        1. $p \xrightarrow{a} p'$ but $q \stackrel{a}{\not\rightarrow}$:
          $stack := PUSH([1, \langle p, q\rangle, a, p'], stack)$;
        2. $p \xrightarrow{a} p'$, $q \xrightarrow{a}$ and $\forall q'$ ($q \xrightarrow{a} q'$ implies $p' \not\sqsubseteq^{k-1}_\Pi q'$) :
          $stack := PUSH([2, \langle p, q\rangle, a, p'], stack)$;
        3. $p \Downarrow a$ but $q \Uparrow a$:
          $stack := PUSH([3, \langle p, q\rangle, a], stack)$;
        4. $p \Downarrow a$, $q \Downarrow a$, $q \xrightarrow{a} q'$ but $p \stackrel{a}{\not\rightarrow}$:
          $stack := PUSH([4, \langle p, q\rangle, a, q'], stack)$;
        5. $p \Downarrow a$, $q \Downarrow a$, $q \xrightarrow{a} q'$, $p \xrightarrow{a}$ and $\forall p'$ ($p \xrightarrow{a} p'$ implies $p' \not\sqsubseteq^{k-1}_\Pi q'$) :
          $stack := PUSH([5, \langle p, q\rangle, a, q'], stack)$;
      end
    end
    $\sqsubseteq^{k+1}_\Pi := \mathcal{F}(\sqsubseteq^k_\Pi)$;
    $k := k + 1$;
  end
  if $p_0 \sqsubseteq^k_\Pi q_0$ **then return** $true$
  else return $(false, stack)$;
end $PREORDER$;

Figure 3: Algorithm $PREORDER$.

Although this idea is straightforward there are some important details to be considered. We recommend the interested reader to refer to [3] for a complete discussion of the algorithm. Note that the formulas generated by DFG for a tuple of the form $[0, \langle p, q\rangle]$ have the form $\Pi_q(p)$. Technically speaking the only atomic formulas involving Π have the form $\Pi(p)$ (see Section 2.2). However it is frequently convenient to have the state q that violates $\Pi(p)$ also returned.

4 Computing Tests

The steps of our method to compute diagnostic tests are as follows:

1. Transform both processes into *acceptance* graphs.
2. Set $\Pi = \{\langle y, z\rangle \mid y.acc = \emptyset \vee (z.acc \neq \emptyset \wedge z.acc \subset\subset y.acc)\}$ (Note that $y.acc = \emptyset$ iff $y.conv$ is false.)

3. Run the $PREORDER$ algorithm on these graphs, and generate a distinguishing formula using DFG if the acceptance graphs are not related by $\sqsubseteq_\Pi$.

4. Transform the resulting formula into a *diagnostic* test using the algorithm $TRANSFORM$ that is given in this section.

Before describing $TRANSFORM$ we first remark on some relevant properties of the formulas generated by DFG when it is invoked on acceptance graphs. Algorithm $PREORDER$ can compute $\sqsubseteq_\Pi$ for any Π and processes $\mathcal{P}, \mathcal{Q}$. However, if the graphs are deterministic (as acceptance graphs are) and $\mathcal{F}$ is computed appropriately, some types of tuples can never be in the stack.

Lemma 4.1 *If the process is deterministic and $\mathcal{F}$ is computed by processing cases in the order from 0 to 5, then*

1. *$PREORDER$ pushes no tuple of type (3) into the stack, and*

2. *$PREORDER$ pushes no tuple of type (5) into the stack.*

Proof.

1. Divergence information is encoded in acceptance sets in the acceptance graphs. So if $p \Downarrow a$ and $q \Uparrow a$ then the tuple $[0, \langle p, q\rangle]$ will be pushed to the stack during initialization because the acceptance sets of p and q are not related by Π.

2. By contradiction. Suppose the graph is deterministic and the cases are processed in the given order. Assume that the stack has a tuple of the form $[5, \langle p, q\rangle, a, p']$. This means that

$$p{\Downarrow}a \wedge q{\Downarrow}a \wedge q \xrightarrow{a} q' \wedge p \xrightarrow{a} \wedge \forall p'. \, (p \xrightarrow{a} p' \wedge p' \not\sqsubseteq_\Pi^{i-1} q')$$

Since the graph is deterministic, there can be only one such p' and q'. This means

$$q \xrightarrow{a} q' \wedge p \xrightarrow{a} p' \wedge p' \not\sqsubseteq_\Pi^{i-1} q'$$

However this is a situation which can be handled by case 2, since condition 2 is fulfilled. Thus, a tuple of type (2) would be pushed into the stack instead.

□

Since the tuples of type (3) or (5) can never exist the work done by DFG is substantially simplified. Algorithm DFG terminates when it pops a tuple of type (0),(1),(3) and (4). Only tuples (2) and (5) causes the algorithm to recurse. As tuples of type (5) are not possible (Lemma 4.1) we have the following result.

Lemma 4.2 *Consider the formula $\Phi = DFG(\langle y, z\rangle, stack)$, where y, z are states in $\mathcal{A}(\mathcal{P}), \mathcal{A}(\mathcal{Q})$ and stack is generated by $PREORDER$.*

1. *Φ does not contain any disjunctions or conjunctions.*

2. *The only subformulas involving $[]_\Downarrow$ are of the form $[a]_\Downarrow ff$.*

Proof.

1. Since the graph is deterministic, every state has at most one $a-$derivative. Therefore we only need to distinguish at most one $a-$derivative of y from at most one $a-$derivative of z. Accordingly, when DFG recurses in handling tuples of type (2), only one "conjunct" will be generated, and no conjunctions are needed as a result. (Note that when z has more than one a-derivative —which is not possible in deterministic graphs— then we may have to generate formulas that distinguish an a-derivative of y from each a-derivative of z and take their conjunction.)

2. If there is a tuple containing $\langle y, z\rangle$ in the stack, then there must be a smallest $k \geq 0$ such that $y \not\sqsubseteq^k_\Pi z$. The proof is based on induction on this k.

   For the base case consider $k = 0$. If $y \not\sqsubseteq^0_\Pi z$ then the tuple in the stack containing y and z is of type (0) and the formula generated is $\Pi_z(y)$ which has no subformula involving $[]_\Downarrow$.

   For the inductive step we will prove that if $y \not\sqsubseteq^{k+1}_\Pi z$ then $DFG(\langle y, z\rangle, stack)$ generates formulas whose subformulas involving $[]_\Downarrow$ are of the form $[a]_\Downarrow ff$. The induction hypothesis states that the result holds for k. The tuple containing y and z can not be of type (3) or (5) because of Lemma 4.1, and tuples of type (0) can not have been pushed into the stack at this stage in the algorithm. Thus there are three cases to consider.

   1. The tuple is of type (1). Then the formula is $\langle a\rangle tt$, which has no subformulas involving $[]_\Downarrow$.
   2. The tuple is of type (2). Then the formula is $\langle a\rangle DFG(\langle y', z'\rangle, stack)$ where $y \xrightarrow{a} y'$ and $z \xrightarrow{a} z'$. Note that there can only be one such z' because the processes are deterministic. By the inductive hypothesis the formula generated by $DFG(\langle y', z'\rangle, stack)$ has the desired form as $y' \not\sqsubseteq^k_\Pi z'$, and thus so does the formula $\langle a\rangle DFG(\langle y, z\rangle, stack)$.
   3. The tuple is of type (4). Then the formula is $[a]_\Downarrow ff$, which is obviously of the right form.

□

Test Generation

Once the formula is computed by DFG the algorithm $TRANSFORM$ transforms it into a *diagnostic* test. It does so by recursing through the formula and applying appropriate transformations. In addition to building a test, $TRANSFORM$ also returns a flag indicating whether the test is a "may" test or a "$must$" test. The idea is that if $\mathcal{P} \not\ll_t \mathcal{Q}$, then it may be that either $\mathcal{P} \not\ll_{may} \mathcal{Q}$ or $\mathcal{P} \not\ll_{must} \mathcal{Q}$. So the flag indicates which preorder is violated, and the test explains why. The relevant pseudo-code appears in Figure 4.

In the following we abuse notation slightly by defining $\mathcal{A}^{-1}(y)$ to be the set of states in $\mathcal{P}$ corresponding to the state y of $\mathcal{A}(\mathcal{P})$. The following theorem states a correspondence between tests generated by $TRANSFORM$ and states in the processes $\mathcal{P}$ and $\mathcal{Q}$.

Theorem 4.3 *Let y and z be two states in the processes $\mathcal{A}(\mathcal{P})$ and $\mathcal{A}(\mathcal{Q})$ such that $y \not\sqsubseteq_\Pi z$. Let $\Phi = DFG(\langle y, z\rangle, stack)$, where stack is built by $PREORDER$.*

(1) If $TRANSFORM(\Phi) = \langle \mathcal{T}, must\rangle$ then $\forall y_i \in \mathcal{A}^{-1}(y)$, y_i must $\mathcal{T}$ but $\exists z_j \in \mathcal{A}^{-1}(z)$ such that z_j m~~u~~st $\mathcal{T}$.

```
TRANSFORM(Φ) → e ;
  case Φ of
    ⟨a⟩Φ'     : ⟨t, flag⟩ = TRANSFORM(Φ') ;
                if flag "may" then return(⟨at, may⟩)
                else return(⟨at + τw, must⟩) ;
    tt        : return(⟨w, may⟩) ;
    Π_z(y)    : return(⟨T(y, z), must)⟩) ;
    [a]⇓ff    : return(⟨a + τw, must⟩) ;
  end case
end TRANSFORM ;

T(y, z) → e ;
  if y.acc ≠ ∅ ∧ z.acc = ∅ then return τw ;
  else \* y.acc ≠ ∅ ∧ z.acc ⊄⊄ y.acc *\
    find one B ∈ z.acc such that ∄A_i ∈ y.acc for which A_i ⊆ B.
    forall A_i ∈ y.acc = {A_1, ..., A_n} do
      pick a_i ∈ A_i − B.
    return (a_1w + a_2w + ... + a_nw) ;
end T ;
```

Figure 4: Algorithm for translating a formula into a test.

(2) *If* $TRANSFORM(\Phi) = \langle \mathcal{T}, may\rangle$ *then* $\exists y_i \in \mathcal{A}^{-1}(y)$ *such that* y_i *may* $\mathcal{T}$ *but* $\forall z_j \in \mathcal{A}^{-1}(z)$, z_j $\cancel{may}$ $\mathcal{T}$.

Proof. We will use $|\Phi|$ to denote the number of $\langle\rangle$ modal operators in Φ. In both cases we prove the desired result by induction on $|\Phi|$. Let t_0 be the start state of $\mathcal{T}$. We start with part 1.

Basis: $|\Phi| = 0$. Since the formula does not contain any $\langle\rangle$'s it can be one of the following two forms: $\Phi = \Pi_z(y)$ or $\Phi = [a]_\Downarrow ff$.

(*i*) $\Phi = \Pi_z(y)$.
In this case, $y.acc \neq \emptyset$. From the definitions of $TRANSFORM$ and T the test $\mathcal{T}$ can have one of two forms.

(a) $\mathcal{T} = \tau w$. This test is generated if $z.acc = \emptyset$. In this case at least one state in $\mathcal{A}^{-1}(z)$ is divergent. Therefore this state may diverge while executing the test τw and prohibit τw from executing its τ and report success. On the other hand, for all states in $\mathcal{A}^{-1}(y)$, $\mathcal{T}$ will eventually be allowed to execute its τ and reporting success, since all $y_i \in \mathcal{A}^{-1}(y)$ must be convergent. So all the states in $\mathcal{A}^{-1}(y)$ *must* τw, but one state in $\mathcal{A}^{-1}(z)$ $\cancel{must}$ τw.

(b) $\mathcal{T} = (a_1 w + a_2 w + \ \dots\ + a_n w)$. This test is computed if $z.acc \neq \emptyset$ and $y \not\Pi z$, which means that $z.acc \not\subset\not\subset y.acc$. In this case there exists a $B \in z.acc$ such that for every $A_i \in y.acc$, $1 \leq i \leq n$, $A_i \not\subseteq B$. Therefore, for each i, there exists some $a_i \in A_i$ for which $a_i \notin B$. Now all the states in $\mathcal{A}^{-1}(y)$ *must* $(a_1 w + a_2 w + \ \dots\ + a_n w)$ because every state in $\mathcal{A}^{-1}(y)$ is capable of doing at least one action from the set $A = \{a_1, a_2, \ldots, a_n\}$ (after potentially performing some finite amount of internal computation first). On the other hand, there is at least one state in $\mathcal{A}^{-1}(z)$ which is only capable of performing actions in B and $B \cap A = \emptyset$. So this state $\not{must}\ \mathcal{T}$.

(ii) $\Phi = [a]_{\Downarrow} ff$.
The test, corresponding to this formula is $a + \tau w$. That formula Φ is a distinguishing formula means that action a is not possible from state y but is possible from state z. However, this implies that no y_i in $\mathcal{A}^{-1}(y)$ can perform a and at least one z_j in $\mathcal{A}^{-1}(z)$ can perform a. Otherwise, the transformation would have created an edge labeled a from y in $\mathcal{A}(\mathcal{P})$, and $[a]_{\Downarrow} ff$ would not have been returned. Therefore, all $y_i \in \mathcal{A}^{-1}(y)$ $must$ $a + \tau w$ because of the fact that since each $y_i \in \mathcal{A}^{-1}(y)$ is convergent, every computation beginning $y_i \| t_0$ (recall that t_0 is the start state of $\mathcal{T}$, which here is $a + \tau w$) must eventually allow the test to perform its τ and report success. On the other hand, at least one state z_k in $\mathcal{A}^{-1}(z)$ is capable of doing action a, and for this reason this z_k $\not{must}$ $a + \tau w$ because of the unsuccessful computation $z_k \| t_0 \xrightarrow{a} z'_k \| NIL$. This proves the basis for case ii.

Induction step: For the induction step, let $n \geq 0$ and assume $|\Phi| = n + 1$. So Φ has the form $\langle b_0 \rangle \Phi'$, where $|\Phi'| = n$. The induction hypothesis states that for any Φ' with $|\Phi'| = n$ that is the result of $DFG(\langle y', z' \rangle, stack)$, if $TRANSFORM(\Phi') = \langle \mathcal{T}', must \rangle$ then $\forall y'_i \in \mathcal{A}^{-1}(y')$ $must$ $\mathcal{T}'$, but $\exists z'_j \in \mathcal{A}^{-1}(z')$ such that z'_j $\not{must}$ $\mathcal{T}'$. We must show this result for Φ. Note that $TRANSFORM(\Phi) = \langle b_0 \mathcal{T}' + \tau w, must \rangle$, meaning $\mathcal{T} = b_0 \mathcal{T}' + \tau w$ and $\langle \mathcal{T}', must \rangle$ is the result of $TRANSFORM(\Phi')$. Assume that t'_0 is the start state of $\mathcal{T}'$. Let y' and z' be the (unique) states such that $y \xrightarrow{b_0} y'$ and $z \xrightarrow{b_0} z'$, respectively, and define Y, Y', Z and Z' as $Y = \mathcal{A}^{-1}(y)$, $Y' = \mathcal{A}^{-1}(y')$, $Z = \mathcal{A}^{-1}(z)$ and $Z' = \mathcal{A}^{-1}(z')$. Although $y \xrightarrow{b_0} y'$ and $z \xrightarrow{b_0} z'$, it is not necessarily the case that all the states in the set Y and all the states in set Z have b_0 transitions.

Now we show that $\forall y_i \in Y$ y_i $must$ $\mathcal{T}$ but $\exists z_j \in Z$ such that z_j $\not{must}$ $\mathcal{T}$. First, consider the states in Y. Now, $y.acc \neq \emptyset$, since otherwise $\langle \mathcal{T}, must \rangle$ could not have been generated. This implies that for all $y_i \in Y$, $y_i \Downarrow$. Now fix $y_i \in Y$; we must show that y_i $must$ $\mathcal{T}$. There are two cases to consider.

1. $y_i \stackrel{b_0}{\not\Rightarrow}$. In this case the only computations are of the form $y_i \| t_0 \xrightarrow{\tau} y_j \| t_0 \xrightarrow{\tau} \ldots$. Since $y_i \Downarrow$, no infinite τ-computations are possible, meaning that during each computation $\mathcal{T}$ is able to execute its τ at some point, after which it is capable of reporting success. Thus each computation beginning with $y_i \| t_0$ is successful and y_i $must$ $\mathcal{T}$.

2. $y_i \stackrel{b_0}{\Rightarrow} y'_k$. From the definition of $\mathcal{A}$ it follows that $y'_k \in Y'$. In addition to the successful computations mentioned above in the case where $y_i \stackrel{b_0}{\not\Rightarrow}$, y_i can exhibit computations of the following form:

$$y_i \| t_0 \stackrel{\epsilon}{\Rightarrow} y_j \| t_0 \xrightarrow{b_0} y'_j \| t'_0 \stackrel{\epsilon}{\Rightarrow} y'_k \| t'_0$$

(From the definition of $+$, t'_0 should in fact be $\langle 0, t'_0 \rangle$. To avoid notational clutter, however, we omit the tags 0 and 1 in what follows.)

From the definition of DFG and $TRANSFORM$, $\Phi' = DFG(\langle y', z' \rangle, stack)$ and $\mathcal{T}'$ is built by $TRANSFORM(\Phi')$; therefore the induction hypothesis guarantees that $y'_k \in$

$\mathcal{A}^{-1}(y')$ *must* $\mathcal{T}'$. Thus all the computations beginning $y_i \| t_0 \overset{\epsilon}{\Rightarrow} y_j \| t_0 \xrightarrow{b_0} y'_j \| t'_0 \overset{\epsilon}{\Rightarrow} y'_k \| t'_0$ are successful as well, and y_i *must* $\mathcal{T}$.

A similar analysis will be done to show that there is at least one state z_j in Z such that z_j $\not{must}$ $\mathcal{T}$. By the induction hypothesis there must exist a z'_k such that z'_k $\not{must}$ $\mathcal{T}'$, i.e. there is an unsuccessful computation beginning with $z'_k \| t'_0$. From the definition of the $\mathcal{A}$ construction, it follows that there must exist $z'_r \in Z'$ and $z_j \in Z$ such that $z_j \xrightarrow{b_0} z'_r \overset{\epsilon}{\Rightarrow} z'_k$. Thus there is an unsuccessful computation beginning

$$z_j \| t_0 \xrightarrow{b_0} z'_j \| t'_0 \overset{\epsilon}{\Rightarrow} z'_k \| t'_0$$

and z_j $\not{must}$ $\mathcal{T}$. This concludes the proof of part 1.

We now prove part 2.

Basis: Each MAY test contains at least one $\langle\rangle$. So our basis is $|\Phi| = 1$. There is only one case to examine, which is $\Phi = \langle b_1 \rangle tt$. The test corresponding to this formula is $\mathcal{T} = b_1 w$. That $\Phi = \langle b_1 \rangle tt$ is a distinguishing formula means action b_1 is possible from y but not from z. However, this implies there is at least one y_i in $\mathcal{A}^{-1}(y)$ such that $y_i \xrightarrow{b_1} y'_i$ for some y'_i and no z_j in $\mathcal{A}^{-1}(z)$ can perform b_1. Therefore y_i *may* $b_1 w$ because of the successful computation

$$y_i \| t_0 \xrightarrow{b_1} y'_i \| t_1,$$

where $t_1 \xrightarrow{w}$, while for all $z_j \in \mathcal{A}^{-1}(z)$, z_j $\not{may}$ $b_1 w$. This completes the proof of the basis case.

Induction step: For the induction step let $n \geq 1$ and $|\Phi| = n + 1$. So Φ has the form $\langle b_0 \rangle \Phi'$ where $|\Phi'| = n$. The induction hypothesis states that for any Φ' with $|\Phi'| = n$ that is the result of $DFG(\langle y', z' \rangle, stack)$, if $TRANSFORM(\Phi') = \langle \mathcal{T}', may \rangle$ then $\exists y'_i \in \mathcal{A}^{-1}(y')$ such that y'_i *may* $\mathcal{T}'$, while $\forall z'_j \in \mathcal{A}^{-1}(z')$ z'_j $\not{may}$ $\mathcal{T}'$. We must show this result for Φ. Note that the test built by $TRANSFORM(\Phi)$ has form $\mathcal{T} = b_0 \mathcal{T}'$ where $TRANSFORM(\Phi') = \langle \mathcal{T}', may \rangle$. Assume t'_0 is the start state of $\mathcal{T}'$. Let y' and z' be the (unique) states such that $y \xrightarrow{b_0} y'$ and $z \xrightarrow{b_0} z'$, respectively, and define Y, Y', Z and Z' as $Y = \mathcal{A}^{-1}(y)$, $Y' = \mathcal{A}^{-1}(y')$, $Z = \mathcal{A}^{-1}(z)$ and $Z' = \mathcal{A}^{-1}(z')$.

Now we show that $\exists y_i \in \mathcal{A}^{-1}(y)$ such that y_i *may* $\mathcal{T}$ but $\forall z_j \in \mathcal{A}^{-1}(z)$ $\not{may}$ $\mathcal{T}$. By the induction hypothesis there must exist a state $y'_i \in Y'$ such that y'_i *may* $\mathcal{T}'$. Also, from the construction of $\mathcal{A}$ there must be a $y_i \in Y$ with $y_i \overset{b_0}{\Rightarrow} y'_i$. Thus $y_i \| t_0$ has a successful computation beginning with $y_i \| t_0 \overset{b_0}{\Rightarrow} y'_i \| t'_0$, and this y_i *may* $\mathcal{T}$.

To prove $\forall z_j \in \mathcal{A}^{-1}(z) z_j$ $\not{may}$ $\mathcal{T}$ fix a z_j in Z. There are two cases to consider.

1. $z_j \overset{b_0}{\not\Rightarrow}$. z_j $\not{may}$ $\mathcal{T}$ because z_j is unable ever to perform b_0.

2. $z_j \overset{b_0}{\Rightarrow}$. By the definition of $\mathcal{A}$, it follows that for every z'_j such that $z_j \overset{b_0}{\Rightarrow} z'_j$, $z'_j \in Z'$. Moreover, by the induction hypothesis, each of these z'_j $\not{may}$ $\mathcal{T}'$. Therefore, each computation beginning with $z_j \| t_0$ must either be $z_j \| t_0 \xrightarrow{\tau} z_k \| t_0 \xrightarrow{\tau} \ldots$ or have the form $z_j \| t_0 \overset{b_0}{\Rightarrow} z'_j \| t'_0$ for some $z'_j \in Z'$, and as these are all unsuccessful, z_j $\not{may}$ $\mathcal{T}$.

□

As a result of this, we may now prove the correctness of $TRANSFORM$.

Theorem 4.4 *Let $\mathcal{P}$ and $\mathcal{Q}$ be two processes. If $\mathcal{P} \not\ll_t \mathcal{Q}$ then $TRANSFORM$ will return a pair $\langle \mathcal{T}, pass \rangle$, with $pass \in \{may, must\}$, such that $\mathcal{P}$ pass $\mathcal{T}$ but $\mathcal{Q}$ $p\!\!\!/ass$ $\mathcal{T}$.*

Proof. Let p_0 and q_0 be the start states of the processes $\mathcal{P}$ and $\mathcal{Q}$, respectively. From the definition of $\mathcal{A}$ it follows that $\mathcal{A}^{-1}(y_0) = (\{p_0\})^\epsilon$ and $\mathcal{A}^{-1}(z_0) = (\{q_0\})^\epsilon$. We consider the cases where $pass = must$ and $pass = may$.

(1) ($must$) Since $\forall y_i \in \mathcal{A}^{-1}(y_0)$ y_i $must$ $\mathcal{T}$ by Theorem 4.3 and $p_0 \in \mathcal{A}^{-1}(y_0)$, p_0 $must$ $\mathcal{T}$. By the definition of $\mathcal{A}$, for every $z_j \in \mathcal{A}^{-1}(z_0)$ $q_0 \stackrel{\epsilon}{\Rightarrow} z_j$. Let z_k be the state in $\mathcal{A}^{-1}(z_0)$ such that z_k $m\!\!\!/ust$ $\mathcal{T}$. Then q_0 $m\!\!\!/ust$ $\mathcal{T}$, since there can be an unsuccessful computation beginning with $q_0 \| t_0 \stackrel{\epsilon}{\Rightarrow} z_k \| t_0$.

(2) (may) By the definition of $\mathcal{A}$, $p_0 \stackrel{\epsilon}{\Rightarrow} y_i$ for any $y_i \in \mathcal{A}^{-1}(y_0)$. By Theorem 4.3 there exists $y_k \in \mathcal{A}^{-1}(y_0)$ with y_k may $\mathcal{T}$. Therefore p_0 may $\mathcal{T}$ because of the successful computation beginning with $p_0 \| t_0 \stackrel{\epsilon}{\Rightarrow} y_k \| t_0$. However, by the same Theorem, $\forall z_j \in \mathcal{A}^{-1}(z_0)$ z_j $m\!\!\!/ay$ $\mathcal{T}$. Since $q_0 \in \mathcal{A}^{-1}(z_0)$, q_0 $m\!\!\!/ay$ $\mathcal{T}$.

□

We now establish the complexity of $TRANSFORM$. In what follows, if $\mathcal{P} = \langle P, Act, \rightarrow, p_0 \rangle$ then $n_\mathcal{P}$ is the number of states, and $m_\mathcal{P}$ is the number of transitions, in $\mathcal{P}$; we also use $|\mathcal{P}|$ to denote $n_\mathcal{P} + m_\mathcal{Q}$.

Theorem 4.5 *Building a* diagnostic *test from a distinguishing formula can be done in $O(n_{\mathcal{A}(\mathcal{P})} n_{\mathcal{A}(\mathcal{Q})} + n_\mathcal{Q} m_\mathcal{P} + n_\mathcal{P} m_\mathcal{Q})$ time.*

Proof. The number of recursive calls of $TRANSFORM$ is bounded by the length of the formula (the length of the formula is defined as the number of $\langle\rangle$ modalities in it), and the number of tuples in the stack is bounded by $n_{\mathcal{A}(\mathcal{P})} n_{\mathcal{A}(\mathcal{Q})}$. Since DFG can be called at most as many times as the number of tuples in the stack and each call generates one $\langle\rangle$, the length of the formula is limited by $n_{\mathcal{A}(\mathcal{P})} n_{\mathcal{A}(\mathcal{Q})}$. When processing a formula of the form $\langle b \rangle \Phi$, $TRANSFORM$ calls itself recursively and then performs $O(1)$ work. When it encounters a formula $\Pi_q(p)$, it calls T. Thus, the time complexity of $TRANSFORM$ is on the order of the sum of the number of $\langle\rangle$ modalities in Φ and the time spent in one call of T. Clearly, the former is $O(n_{\mathcal{A}(\mathcal{P})} n_{\mathcal{A}(\mathcal{Q})})$.

Now consider T; the time spent in this procedure is bounded by $O(n_\mathcal{Q} m_\mathcal{P} + n_\mathcal{P} m_\mathcal{Q})$. This follows from the fact that, in the worst case, $|y.acc|$ and $|z.acc|$ are bounded by $n_\mathcal{P}$ and $n_\mathcal{Q}$, respectively, with each set S in $y.acc$ and S' in $z.acc$ corresponding to a state p in P and a state q in Q, respectively. As a result $|S|$ and $|S'|$ are bounded by $|init(p)|$ and $|init(q)|$. Therefore, finding an $S' \in y.acc$ such that $\forall S \in y.acc$ $S' \not\subseteq S$ takes time proportional to

$$\begin{aligned} C(z.acc \not\subset\not\subset y.acc) &= \sum_{p \in P} \sum_{q \in Q} (|init(p)| + |init(q)|) \\ &\leq \sum_{p \in P} (n_\mathcal{Q} |init(p)| + m_\mathcal{Q}) \\ &\leq n_\mathcal{Q} m_\mathcal{P} + n_\mathcal{P} m_\mathcal{Q} \end{aligned}$$

Accordingly, building a test from a formula can be done in $O(n_{\mathcal{A}(\mathcal{P})} n_{\mathcal{A}(\mathcal{Q})} + n_\mathcal{Q} m_\mathcal{P} + n_\mathcal{P} m_\mathcal{Q})$ time.
□

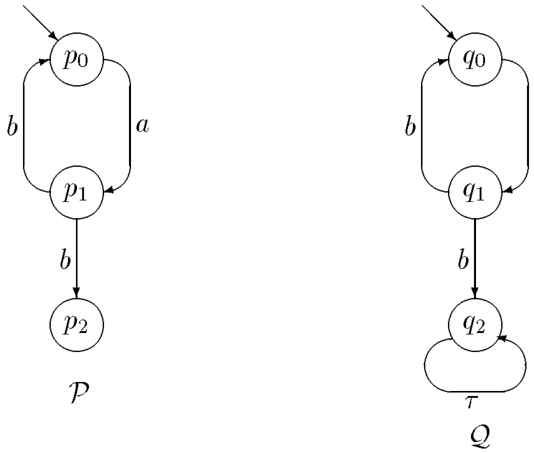

Figure 5: Two processes.

Note that building a test does not affect the complexity of computing $<<_t$. Computing Π takes

$$\begin{aligned} C(\Pi) &= \sum_{\langle y,z\rangle \in Y\times Z} C(z.acc \not\subset\!\!\not\subset y.acc) \\ &\leq n_{\mathcal{A}(\mathcal{P})} n_{\mathcal{A}(\mathcal{Q})} (n_{\mathcal{Q}} m_{\mathcal{P}} + n_{\mathcal{P}} m_{\mathcal{Q}}). \end{aligned}$$

where Y and Z are the set of states in the acceptance graphs of $\mathcal{P}$ and $\mathcal{Q}$ respectively. However, from [5] it follows that $\sqsubseteq\!\!\!\!\sim$ may be computed in $C(\sqsubseteq\!\!\!\!\sim) = O(|\mathcal{P}| \cdot |\mathcal{Q}|)$ for any processes $\mathcal{P}$ and $\mathcal{Q}$, implying that $<<_t$ can be computed in time

$$C(\sqsubseteq\!\!\!\!\sim) + C(\Pi) = O(|\mathcal{A}(\mathcal{P})| \cdot |\mathcal{A}(\mathcal{Q})| + n_{\mathcal{A}(\mathcal{P})} n_{\mathcal{A}(\mathcal{Q})} (n_{\mathcal{Q}} m_{\mathcal{P}} + n_{\mathcal{P}} m_{\mathcal{Q}})).$$

In general $|\mathcal{A}(\mathcal{P})|$ can be exponential in $|\mathcal{P}|$, owing to the well-known explosion in the number of states that can result from determinizing of a finite-state machine. In practice, however, $|\mathcal{A}(\mathcal{P})|$ is usually *smaller* than $|\mathcal{P}|$, owing to the collapsing of τ-transitions.

5 Examples

In this section we present two examples to illustrate our method. For the first example we will use the two processes shown in Figure 5. In this case the left process $\not<<_t$ the right process. State p_0 is the start state of one process, and state q_0 is the start state of the other. The corresponding acceptance graphs are shown in Figure 6. Note that these graphs are slightly different from those that would be created by the traditional determinization algorithms. This is because the encoding of divergence information into the graphs may yield two states where the traditional algorithms would only build one. Recall that the states that have $\emptyset$ as their acceptance sets are divergent states.

After this transformation, the preorder algorithm is executed on these graphs, and the stack is formed. Then the formula-generation algorithm builds the distinguishing formula using the information in the stack. In this case the stack contains the elements $[2, \langle y_0, z_0\rangle, a, y_1]$, $[2, \langle y_1, z_1\rangle, b, y_2]$

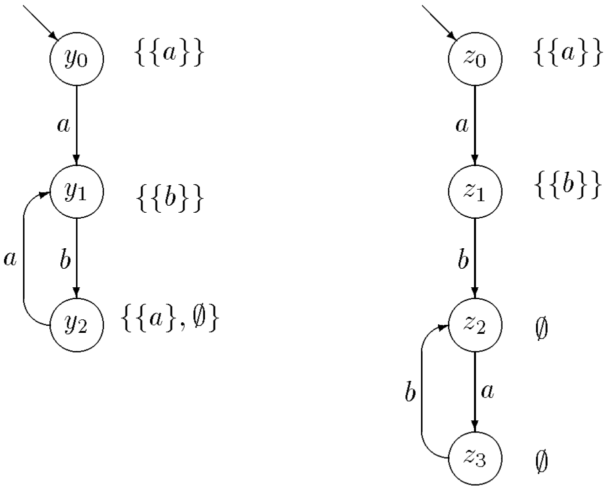

Figure 6: Acceptance graphs.

and $[0, \langle y_2, z_2 \rangle]$ ordered from top to bottom, respectively. DFG then returns the formula:

$$DFG(y_0, z_0) = \langle a \rangle \langle b \rangle \Pi_{z_2}(y_2)$$

This formula holds of y_0 and not z_0. Note that this formula is not directly relevant to the original graphs $\mathcal{P}$ and $\mathcal{Q}$.

Now we compute a test which distinguishes the original testing inequivalent graphs on the basis of this formula. The algorithm $TRANSFORM$ calls itself recursively, each time eliminating one $\langle\ \rangle$ until it locates the atomic formula $\Pi_{z_2}(y_2)$. Then it calls T to compute the test corresponding to this atomic formula. After this, it recursively returns the test and the type of the test (*may* or *must*). In this case the test for the formula $\Pi_{z_2}(y_2)$ is τw and the type of the test is *must*. The execution sequence of the algorithm for the formula $\langle a \rangle \langle b \rangle \Pi_{z_2}(y_2)$ is

$$\begin{array}{l}
TRANSFORM(\langle a \rangle \langle b \rangle \Pi_{z_2}(y_2)) \\
\qquad TRANSFORM(\langle b \rangle \Pi_{z_2}(y_2)) \\
\qquad\qquad TRANSFORM(\Pi_{z_2}(y_2)) \\
\qquad\qquad\qquad T(y_2, z_2) \rightarrow \tau w \\
\qquad\qquad \langle \tau w, must \rangle \\
\qquad \langle b(\tau w) + \tau w, must \rangle \\
\langle a(b(\tau w) + \tau w) + \tau w, must \rangle
\end{array}$$

So the test distinguishing p_0 from q_0 is

$$a(b\tau w + \tau w) + \tau w.$$

Since this is a *must* test p_0 can pass it only if every computation on the test $a(b\tau w + \tau w) + \tau w$ leads to success. This is indeed the case because the following computations are successful.

$$p_0 \| a(b\tau w + \tau w) + \tau w \xrightarrow{a} p_1 \| b\tau w + \tau w \xrightarrow{b} p_2 \| \tau w \xrightarrow{\tau} p_2 \| w$$

```
process PROTOCOL[send,rec] :=
   hide link in
      SENDER[send,link] |[link]| RECEIVER[link,rec]
   where
      process SENDER[s,l] :=
         s; l; SENDER[s,l]
      endproc
      process RECEIVER[l,r] :=
         l; r; RECEIVER[l,r]
      endproc
endproc

specification SERVICE[send,rec] :=
   behaviour
      send; rec; SERVICE[send,rec]
endspec
```

Figure 7: Sender-Receiver system.

$$p_0||a(b\tau w + \tau w) + \tau w \xrightarrow{a} p_1||b\tau w + \tau w \xrightarrow{b} p_0||\tau w \xrightarrow{\tau} p_0||w$$

$$p_0||a(b\tau w + \tau w) + \tau w \xrightarrow{a} p_1||b\tau w + \tau w \xrightarrow{\tau} p_1||w$$

$$p_0||a(b\tau w + \tau w) + \tau w \xrightarrow{\tau} p_0||w$$

On the other hand, q_0 fails while performing this test because of the unsuccessful computation

$$q_0||a(b\tau w + \tau w) + \tau w \xrightarrow{a} q_1||b\tau w + \tau w \xrightarrow{b} q_2||\tau w \xrightarrow{\tau} q_2||\tau w \xrightarrow{\tau} \ldots$$

State q_2 can continuously perform $\tau-$actions, and the test can thus be prevented from reporting success.

The second example defines a very simple protocol consisting of two processes: a sender that sends messages through a link and a receiver that receives these messages from this link. The service specification for this protocol requires that each message sent by the sender is received by the receiver before another message is sent. The descriptions of the protocol and service specifications in LOTOS [2] appear in Figure 7.

The corresponding transition graphs and acceptance graphs are shown in Figure 8 and Figure 9, respectively. An action through the hidden gate *link* is represented by τ. After the preorder algorithm is executed the stack contains the elements $[2, \langle y_0, z_0\rangle, send, y_1]$ and $[4, \langle y_1, z_1\rangle, send, z_2]$ ordered from top to bottom respectively. DFG then returns the formula :

$$DFG(y_0, z_0) = \langle send\rangle[send]_{\Downarrow} ff$$

and TRANSFORM returns the test :

$$send(send + \tau w) + \tau w$$

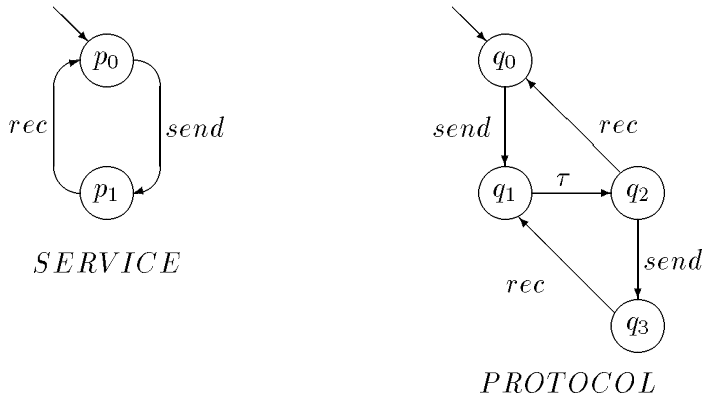

Figure 8: Transition graphs for Sender-Receiver system.

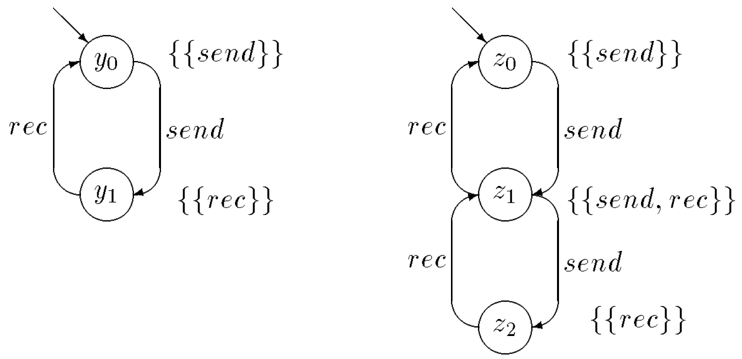

Figure 9: Acceptance graphs for Sender-Receiver system.

$SERVICE$ passes this test because every computation leads to success. However, $PROTOCOL$ fails on this test because of the unsuccessful computation :

$$q_0 \| send(send + \tau w) + \tau w \xrightarrow{send} q_1 \| send + \tau w \xrightarrow{\tau} q_2 \| send + \tau w \xrightarrow{send} q_3 \| nil$$

The service specification does not allow two consecutive send operations.

6 Conclusions

In this paper we have shown how to generate tests that distinguish one process from another process that fails to be higher than it in the *testing preorder*. Such tests serve a useful diagnostic role in system verification; if a system fails to be larger than its specification in the testing preorder, such a test can be seen as an explanation of why. Our method relies on a "postprocessor" to translate diagnostic information generated by a generic preorder checker into appropriate tests. The complexity of the postprocessing step does not affect the complexity of the $<<_t$ algorithm. As future work we plan to incorporate this test generation capability into the Concurrency Workbench [4], a tool for the analysis of finite-state systems.

References

[1] Bolognesi, T. and M. Caneve, **"Squiggles: A Tool for the Analysis of LOTOS Specifications"**, *Formal Description Techniques*, K.J. Turner ed., pp. 201-216. Elsevier Science Publishers B.V., Amsterdam, 1989.

[2] Bolognesi, T. and E. Brinksma, **"Introduction to the ISO Specification Language LOTOS"**, *Computer Networks and ISDN Systems,* vol. 14, (1987), 25-59.

[3] Celikkan, U. and R. Cleaveland, **"On Computing Diagnostic Information for Preorder Checking"**, To appear in International Workshop on Computer-Aided Verification.

[4] Cleaveland, R., J. Parrow, and B. Steffen, **" The Concurrency Workbench"**, In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems,* LNCS 407, (1989), 24-37. To appear in *ACM TOPLAS.*

[5] Cleaveland, R., and B. Steffen, **" Computing Behavioral Relations, Logically"**, In *Proceedings of ICALP,* (1991).

[6] Cleaveland, R., and M. Hennessy, **"Testing Equivalence as a Bisimulation Equivalence"**, In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems,* LNCS 407, (1989), 11-23. To appear in *Fundamental Aspects of Computing.*

[7] DeNicola, R., and M.C.B. Hennessy, **"Testing Equivalences for Processes"**, *Theoretical Computer Science,* vol. 24, (1984), 83-113.

[8] Hennessy, M., *Algebraic Theory of Processes*, MIT Press, 1988.

[9] Hennessy, M., and R. Milner, **"Algebraic Laws for Nondeterminism and Concurrency"**, *Journal of the Association for Computing Machinery,* vol. 32, no. 1, (January 1985), 147-161.

[10] Milner, R., *Communication and Concurrency*, Prentice Hall, 1989.

[11] Stirling, C., **"Modal Logics for Communicating Systems"**, *Theoretical Computer Science,* vol. 49, (1987), 311-347.

[12] Walker, D., **"Bisimulations and Divergence"**, In *Proceedings of the Third Annual Symposium on Logic in Computer Science,* (1988), 186-192.

Appendix

```
DFG (⟨p, q⟩, stack) → Φ ;
    tuple := TOP(stack) ;
    stack := POP(stack) ;
    if ⟨p, q⟩ not in tuple then DFG(⟨p, q⟩, stack) ;
    else
        Γ := ∅ ;
        case tuple of
          [0, ⟨p, q⟩]          : return Π_q(p) ;
          [1, ⟨p, q⟩, a, p′]   : return ⟨a⟩tt ;
          [2, ⟨p, q⟩, a, p′]   : R = { s′ | q -a-> s′ } ;
                                 foreach s′ ∈ R do
                                     Φ′ = DFG(⟨p′, s′⟩, stack) ;
                                     Γ = Γ ∪ {Φ′} ;
                                 enddo
                                 return ⟨a⟩(∧Γ) ;
          [3, ⟨p, q⟩, a]       : return [a]⇓tt ;
          [4, ⟨p, q⟩, a, q′]   : return [a]⇓ff ;
          [5, ⟨p, q⟩, a, q′]   : R = { s′ | p -a-> s′ } ;
                                 foreach s′ ∈ R do
                                     Φ′ = DFG(⟨s′, q′⟩, stack) ;
                                     Γ = Γ ∪ {Φ′} ;
                                 enddo
                                 return [a]⇓(∨Γ);
        endcase
    endelse
end DFG ;
```

Figure 10: Code for computing distinguishing formulas.