# Generating Diagnostic Information for Behavioral Preorders*

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 describes a method for generating diagnostic information for the prebisimulation preorder. This information takes the form of a logical formula explaining why a particular process is not larger than the other in the preorder. Our method relies on modifying an algorithm for computing the prebisimulation preorder to save the information needed for generating these distinguishing formulas. As a number of other behavioral preorders may be characterized in terms of prebisimulation preorder, our technique may be used as a basis for computing diagnostic information for these preorders as well.

## 1 Introduction

Research in the area of process algebras has sparked interest in *behavioral relations* as tools for verifying processes [15, 18, 20]. In one approach, one uses 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. These relations have not received as much attention in the literature as behavioral equivalences ([18, 2, 4, 16, 12, 15]) but they are very useful in that the partiality that is allowed in specifications gives system implementors greater flexibility in developing correct implementations. This partiality can also be exploited when developing specifications for components that are to be used in particular network contexts [10, 17, 20], since the constraints that the rest of the network places on the component typically permit many different (and inequivalent) implementations to render the desired behavior of the over-all system. At least two automated tools [8, 13] include algorithms for computing certain preorders over finite-state processes.

Our goal in this paper is to develop an algorithm for generating diagnostic information for a particular preorder, the *prebisimulation preorder* [1, 20]; the algorithm is to be used in conjunction with a method for computing the preorder to generate information explaining why a particular process is *not* larger than another. This information may then be used by system designers to analyze why systems fail to meet their (partial) specifications. The prebisimulation preorder

* Research supported by NSF/DARPA research grant CCR-9014775.

is of interest in its own right; moreover, it may be used as a basis for calculating other behavioral preorders such as trace containment (also known as the *may preorder* [12, 15]), the simulation preorder and the testing/failures preorder [7, 15]. It also has a *logical* characterization: there is a simple modal logic having the property that one process is less than another in the preorder exactly when each formula satisfied by the first process is also satisfied by the second. Thus, when a process is *not* less than another, there exists a formula satisfied by the first and not the second. Our algorithm builds such a formula, and it does so without affecting the complexity of the preorder algorithm.

The remainder of the paper is structured as follows. The next section develops our process model and reviews the definition of the prebisimulation preorder and its logical characterization. Section 3 presents a particular algorithm for computing the preorder, and then Section 4 presents our method for generating diagnostic information. Section 5 gives an example illustrating our technique. Section 6 shows how the method may be used to generate diagnostic information for preorders other than the prebisimulation preorder, while the last section contains our conclusions and directions for future research.

## 2 Processes, Preorders, and Intuitionistic Hennessy - Milner Logic

### 2.1 Transition Systems

We use *extended labeled transition systems* to model process behavior. These bear a certain resemblance to nondeterministic finite state automata; to define them, we first introduce the more familiar notion of labeled transition system.

**Definition 1.** A *Labeled Transition System* (lts) is a triple $\langle P, Act, \rightarrow\rangle$ where

1. $P$ is a set of *states*;
2. $Act$ is set of *actions* containing a distinguished *silent* action $\tau$; and
3. $\rightarrow \subseteq P \times Act \times P$ is the *transition relation*.

The intuitive meaning of these components is as follows. $P$ represents the set of possible computation states, $Act$ contains the actions that computations may consist of, and $\rightarrow$ describes the state transitions that may result from the execution of an action in a state. For convenience we use the notation $p \xrightarrow{a} p'$ in place of $(p, a, p') \in \rightarrow$ and read it as $p$ performs $a$ and becomes $p'$. When $p \xrightarrow{a} p'$ holds, we often refer to $p'$ as an $a$-derivative of $p$. We also write $p \xrightarrow{a}$ if there is a $p'$ such that $p \xrightarrow{a} p'$. The action $\tau$ represents an *internal* computation step.

Extended labeled transition systems are then labeled transition systems in which states may be designated as underdefined.

**Definition 2.** An *Extended Labeled Transition System* (elts) is a quadruple $\langle P, Act, \rightarrow, \uparrow\rangle$ where the triple $\langle P, Act, \rightarrow\rangle$ is a lts and $\uparrow \subseteq P \times Act$ is the undefinedness relation.

The relation $\uparrow$ represents a notion of *underdefinedness* or *incomplete* description. If $(p, a) \in \uparrow$ then the behavior of $p$ in response to action $a$ may not be fully given yet; other $a$-transitions might be added in later. We shall use $p \uparrow a$ in place of $(p, a) \in \uparrow$ and write $p \downarrow a$ in lieu of $\neg(p \uparrow a)$. For historical reasons, when $p \uparrow a$ holds we sometimes say that $p$ is $a$-divergent, and when $p \downarrow a$ holds we say $p$ is $a$-convergent. In the sequel we only consider finite-state *elts*. An *elts* is finite-state if $|P| < \infty$ and $|Act| < \infty$. Note that these imply the finiteness of $| \rightarrow |$ and $| \uparrow |$.

Given an extended labeled transition system, processes may be defined by identifying a distinguished state as the start state. Formally a process is a pair $(\langle P, Act, \rightarrow, \uparrow\rangle, p_0)$ where $\langle P, Act, \rightarrow, \uparrow\rangle$ is an extended labeled transition system and $p_0 \in P$ is the start state. Figure 1 shows such a process.

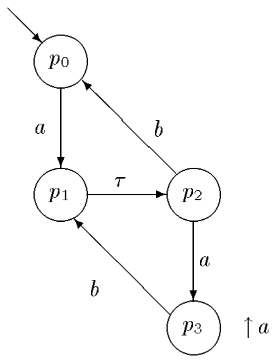


**Fig. 1.** An extended labeled transition system.

Let $\langle P, Act, \rightarrow\rangle$ be a *lts*. There is a well-known method for generating an *elts* from such a *lts*. The construction is designed to abstract away from the internal computation such a system engages in by introducing a new transition relation $\Rightarrow$ and a divergence relation $\Uparrow$ to support this abstraction. Formally we may define the following:

- $\stackrel{\epsilon}{\Rightarrow}$ if $p_1(\stackrel{\tau}{\rightarrow})^n p_2$, for some $n \geq 0$. (so $p_1$ may do $n$ $\tau$-steps and evolve to $p_2$.)
- $p_1 \stackrel{a}{\Rightarrow} p_2$ if there are $p_1', p_2'$ such that $p_1 \stackrel{\epsilon}{\Rightarrow} p_1' \stackrel{a}{\rightarrow} p_2' \stackrel{\epsilon}{\Rightarrow} p_2$.
- $p_1 \Uparrow$ iff there is an infinite sequence $\langle p_i \mid i \geq 1\rangle$ with $p_i \stackrel{\tau}{\rightarrow} p_{i+1}$ for all $i \geq 1$.
- $p_1 \Uparrow a$ iff either $p \Uparrow$ or for some $p'$, $p \stackrel{a}{\Rightarrow} p'$ and $p' \Uparrow$.

Intuitively, $p \stackrel{a}{\Rightarrow} p'$ holds if from $p$ some internal computation may lead to a state in which an $a$ is performed with additional internal computation then leading to $p'$. If $p \Uparrow a$ holds, then $p$ may be triggered by means of an $a$ action into an infinite internal computation. $\langle P, (Act - \{\tau\}) \cup \{\epsilon\}, \Rightarrow, \Uparrow\rangle$ is the extended labeled transition system considered, for example, by [20].

Labeled transition systems provide a flexible basis for reasoning in a number of different specification formalisms like CSP [16], CCS [18]and LOTOS [3]. Finite-state concurrent systems specified in those formalisms can be analyzed automatically once they are converted into labeled transition systems as there exists well-known techniques to study them.

### 2.2 Prebisimulation Preorder

The prebisimulation preorder, $\sqsubseteq\!\!\!\!\sim$, is a behavioral preorder defined in terms of prebisimulations [20]; it is a reflexive and transitive relation that relates processes using transitions and divergence information. Under this relation divergent programs approximate similar convergent ones. 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 behavior of $p$ with respect to $a$ is only partially specified, then $p$ is not required to match the $a$-transitions of $q$. Intuitively this is because as a result of "completing" $p$ with respect to $a$, additional $a$ transitions may be added that could match $q$'s $a$-transitions. The formal definition may be given as follows.

**Definition 3.** Let $\mathcal{P} = (\langle P, Act, \rightarrow, \uparrow\rangle, p_0)$ and $\mathcal{Q} = (\langle Q, Act, \rightarrow, \uparrow\rangle, q_0)$ be processes. A relation $R \subseteq P \times Q$ is a prebisimulation between $\mathcal{P}$ and $\mathcal{Q}$ if $pRq$ implies the following.

1. $p \xrightarrow{a} p' \Rightarrow \exists q'.\ q \xrightarrow{a} q' \wedge p'Rq'$.
2. $p \downarrow a \Rightarrow [q \downarrow a \wedge (q \xrightarrow{a} q' \Rightarrow \exists p'.\ p \xrightarrow{a} p' \wedge p'Rq')]$.

We say that $\mathcal{P} \sqsubseteq\!\!\!\!\sim \mathcal{Q}$ if there is a prebisimulation $R$ with $p_0 R q_0$.

There is a close connection between $\sqsubseteq\!\!\!\!\sim$ and the relation $\sqsubseteq\!\!\!\!\sim_W$ studied by Walker [20]. Suppose $\mathcal{P}, \mathcal{Q}$ are of the form $\langle M, p_0\rangle$ and $\langle N, q_0\rangle$, where $M, N$ are labeled transition systems. Let $\mathcal{P}', \mathcal{Q}'$ be computed from $\mathcal{P}, \mathcal{Q}$ respectively by replacing $\Rightarrow$ for $\rightarrow$ and generating $\Uparrow$ from $\xrightarrow{\tau}$ using the construction specified in Section 1. Then $\mathcal{P}' \sqsubseteq\!\!\!\!\sim \mathcal{Q}'$ iff $\mathcal{P} \sqsubseteq\!\!\!\!\sim_W \mathcal{Q}$.

The prebisimulation preorder may also be used as a basis for computing preorders other than $\sqsubseteq\!\!\!\!\sim_W$. To do so, we slightly refine the usual notion of prebisimulation by introducing a "compatibility relation", $\Pi$, between states of $\mathcal{P}$ and $\mathcal{Q}$. Sometimes, states contain information in addition to their outgoing transitions that is of interest, and $\Pi$ determines when this extra information in two states is compatible. Then a relation $R$ is a $\Pi$-prebisimulation if $R \subseteq \Pi$ and $R$ is a prebisimulation. So if $pRq$, then $p$ and $q$ must be related by $\Pi$ in addition to having their transitions matched appropriately by $R$.

### 2.3 Intuitionistic Hennessy-Milner Logic

The prebisimulation preorder also has a logical characterization in terms of Intuitionistic Hennessy-Milner Logic (IHML) [19]. The syntax of formulae in IHML is defined as follows, where $a \in Act$:

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

The formal semantics of IHML is given in terms of a satisfaction relation, $\models$, relating states in a process $\mathcal{P} = (\langle P, Act, \rightarrow, \uparrow \rangle, p_0)$ to formulas. Formally $\models$ is defined to be the smallest relation satisfying the following, where $p \in P$:

$p \models tt$
$p \models \Phi_1 \wedge \Phi_2$ if $p \models \Phi_1$ and $p \models \Phi_2$
$p \models \Phi_1 \vee \Phi_2$ if $p \models \Phi_1$ or $p \models \Phi_2$
$p \models \langle a \rangle \Phi$ 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 $p_0 \models \Phi$.

IHML incorporates divergence sensitivity into classical Hennessy-Milner Logic (HML) [14]. The chief difference between IHML and HML is that modal operators $\langle a \rangle$ and $[a]_{\downarrow}$ are not duals of each other; for $p$ to satisfy $[a]_{\downarrow}$ it must be completely defined with respect to action $a$. This requirement reflects the intuition that if $p \uparrow a$ , then more $a$-transitions may be added to $p$ later. Thus we can only make statements about all of $p$'s $a$-transitions if we know they have all been given.

The logical characterization of $\precsim$ states that if $\mathcal{P} \precsim \mathcal{Q}$ then the set of formulas satisfied by $\mathcal{P}$ is a subset of $\mathcal{Q}$'s, although $\mathcal{Q}$ may satisfy extra formulas [19]. Formally let $H(\mathcal{P})$ be the set of IHML formulas that a process $\mathcal{P}$ satisfies:

$$H(\mathcal{P}) = \{ \Phi \mid \mathcal{P} \models \Phi \}$$

The next theorem is due to Stirling [19].

**Theorem 4.** *$H(\mathcal{P}) \subseteq H(\mathcal{Q})$ iff $\mathcal{P} \precsim \mathcal{Q}$.*

This theorem suggests that when $\mathcal{P} \not\precsim \mathcal{Q}$, then there is a formula $\Phi$ with $\mathcal{P} \models \Phi$ and $\mathcal{Q} \not\models \Phi$. Thus one way of explaining why $\mathcal{P} \not\precsim \mathcal{Q}$ is to exhibit such a formula.

**Definition 5.** Let $\mathcal{P} = (\langle P, Act, \rightarrow, \uparrow \rangle, p_0)$ and $\mathcal{Q} = (\langle Q, Act, \rightarrow, \uparrow \rangle, q_0)$ be two processes. IHML formula $\Phi$ distinguishes $\mathcal{P}$ from $\mathcal{Q}$ if $\mathcal{P} \models \Phi$ and $\mathcal{Q} \not\models \Phi$.

As an example, consider the two processes $\mathcal{P}$ and $\mathcal{Q}$ given in Figure 2. The formula $\Phi = \langle a \rangle (\langle b \rangle tt \wedge \langle c \rangle tt)$ distinguishes $\mathcal{P}$ from $\mathcal{Q}$; $\mathcal{P} \models \Phi$ but $\mathcal{Q} \not\models \Phi$. Process $\mathcal{Q}$ after performing action $a$ can only do $b$ or $c$ but not both. On the other hand $\mathcal{P}$ can do both $b$ and $c$ after $a$. Note that the existence of this $\Phi$ implies that $\mathcal{P} \not\precsim \mathcal{Q}$.

## 3 Computing the Preorder

The prebisimulation preorder has an iterative characterization that makes it suitable for algorithmic computation. The algorithm we are about to present is based on this characterization, which is as follows:

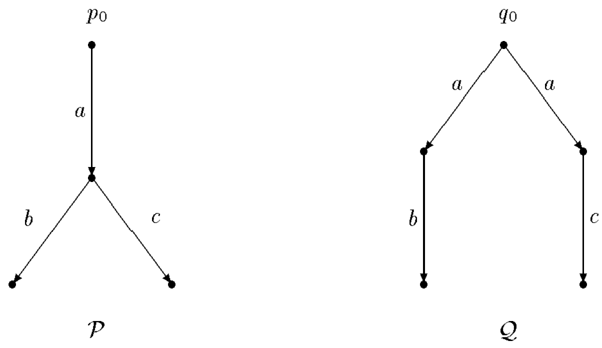


**Fig. 2.** Example Processes (with $\uparrow = \emptyset$)

**Definition 6.** Let $\mathcal{P} = (\langle P, Act, \rightarrow, \uparrow\rangle, p_0)$ and $\mathcal{Q} = (\langle Q, Act, \rightarrow, \uparrow\rangle, q_0)$ be processes. Then a family of relations $\underset{\sim}{\sqsubset}_k \subseteq P \times Q$ can be defined as follows:

1. $\underset{\sim}{\sqsubset}_0 = P \times Q$.
2. $\underset{\sim}{\sqsubset}_{k+1} = \mathcal{F}(\underset{\sim}{\sqsubset}_k)$ where $\mathcal{F}(R) = \{\langle p, q\rangle \mid$
   $(p \xrightarrow{a} p' \Rightarrow \exists q'.\, q \xrightarrow{a} q' \wedge p' R q') \wedge$
   $(p \downarrow a \Rightarrow [q \downarrow a \wedge (q \xrightarrow{a} q' \Rightarrow \exists p'.\, p \xrightarrow{a} p' \wedge p' R q')])\}$.

Note that if $R$ is a prebisimulation then $R \subseteq \mathcal{F}(R)$. We now have the following.

**Theorem 7.** *Let $\mathcal{P}, \mathcal{Q}$ be processes with start states $p_0$ and $q_0$ respectively. Then $\mathcal{P} \underset{\sim}{\sqsubset} \mathcal{Q}$ iff $\forall\, k \quad p_0 \underset{\sim}{\sqsubset}_k q_0$.*

Thus on the basis of this characterization $\underset{\sim}{\sqsubset}$ can be computed by repeatedly applying $\mathcal{F}$ to $\underset{\sim}{\sqsubset}_0$ until $\underset{\sim}{\sqsubset}_{k+1} = \underset{\sim}{\sqsubset}_k$. This suggests the algorithm *PREORDER* (Figure 3) for computing $\underset{\sim}{\sqsubset}$.

## 4 Generating Diagnostic Information

From the definition $\mathcal{F}$ it follows that if $p \not\underset{\sim}{\sqsubset}_k q$ then either $p \not\underset{\sim}{\sqsubset}_{k-1} q$ is also true, or one of the three conditions below must hold.

1. $p \xrightarrow{a} p'$ and $\forall q'(q \xrightarrow{a} q'$ implies $p' \not\underset{\sim}{\sqsubset}_{k-1} q')$
2. $p \downarrow a$ but $q \uparrow a$
3. $p \downarrow a$, $q \downarrow a$, $q \xrightarrow{a} q'$ and $\forall p'(p \xrightarrow{a} p'$ implies $p' \not\underset{\sim}{\sqsubset}_{k-1} q')$

This observation leads to a two-step procedure for the computation of diagnostic information. In the first step, when two states are found not to be related by $\underset{\sim}{\sqsubset}_k$ the information as to which condition is violated and why is collected, encoded as tuples and then pushed onto a stack. These tuples have the following format.

```
k := 1 ;
```

$k := 1$ ;
$\precsim_0 := P \times Q$;
$\precsim_1 := \mathcal{F}(\precsim_0)$;
**while** $\precsim_k \neq \precsim_{k-1}$ **do**
  $\precsim_{k+1} := \mathcal{F}(\precsim_k)$;
  $k := k + 1$ ;
**end**
**if** $p_0 \precsim_k q_0$ **then return** *true*
**else return** *false*;

**Fig. 3.** Algorithm *PREORDER* for computing preorder.

1) $[1, \langle p, q\rangle, a, p']$
2) $[2, \langle p, q\rangle, a]$
3) $[3, \langle p, q\rangle, a, q']$

The tag in the tuple corresponds to the condition that is violated. For example, if $p \downarrow a$ but $q \uparrow a$ then $[2, \langle p, q\rangle, a]$ will be pushed into stack, and if $p \xrightarrow{a} p'$ but $q \stackrel{a}{\not\rightarrow}$ then $[1, \langle p, q\rangle, a, p']$ will be pushed into the stack. Figure 4 contains a version of *PREORDER* in which these tuples are generated and stacked during the computation of the $\precsim_k$.

In the second step, using the information pushed onto the stack in the first step, a distinguishing formula is generated which is satisfied by $\mathcal{P}$ but not by $\mathcal{Q}$.

We now remark on some properties that hold of tuples that the modified *PREORDER* pushes into its stack.

**Theorem 8.**
1. *When* $[1, \langle p, q\rangle, a, p']$ *is pushed onto the stack then it follows that for all* $q'$ *such that* $q \xrightarrow{a} q'$, *a tuple containing* $\langle p', q'\rangle$ *is already in the stack.*
2. *When* $[3, \langle p, q\rangle, a, q']$ *is pushed onto the stack then it follows that for all* $p'$ *such that* $p \xrightarrow{a} p'$, *a tuple containing* $\langle p', q'\rangle$ *is already in the stack.*

*Proof.* $\precsim_{k-1}$ is always computed before $\precsim_k$. So all the pairs which are $\not\precsim_{k-1}$ are pushed onto the stack before those of $\not\precsim_k$.

It also follows that if $p \not\precsim q$ then the tuple containing $\langle p, q\rangle$ is also in the stack when *PREORDER* terminates.

We now remark on the time and space complexity of *PREORDER*.

**Theorem 9.**
1. *The time complexity of the algorithm PREORDER is* $O(|P|^2 \times |Q|^2 \times max(|{\rightarrow_P}|, |{\rightarrow_Q}|))$
2. *The space complexity of the algorithm PREORDER is* $O(|P| \times |Q| + |{\rightarrow_P}| + |{\rightarrow_Q}|)$

```
PREORDER(𝒫 : (⟨P, Act, →⟩, p₀); 𝒬 : (⟨Q, Act, →⟩, q₀)); → stack;
  k := 1;
  ⊑₀ := P × Q;
  ⊑₁ := ℱ(⊑₀);
  while ⊑ₖ ≢ ⊑ₖ₋₁ do
    foreach ⟨p, q⟩ such that p ⊑ₖ₋₁ q but p ⋢ₖ q do
      case condition of
        1. p -a-> p' and ∀q' (q -a-> q' implies p' ⋢ₖ₋₁ q') :
             stack := PUSH([1, ⟨p, q⟩, a, p'], stack)
        2. p↓a but q↑a :
             stack := PUSH([2, ⟨p, q⟩, a], stack)
        3. p↓a, q↓a, q -a-> q' and ∀p' (p -a-> p' implies p' ⋢ₖ₋₁ q') :
             stack := PUSH([3, ⟨p, q⟩, a, q'], stack)
      end
    end
    ⊑ₖ₊₁ := ℱ(⊑ₖ);
    k := k + 1;
  end
  if p₀ ⊑ₖ q₀ then return true
  else return (false, stack);
end PREORDER;
```

**Fig. 4.** Modified *PREORDER*.

It should be noted that these bounds can be significantly improved; for example, an $O(|\mathcal{P}| \times |\mathcal{Q}|)$ algorithm to compute the preorder is given in [11]. Our procedure for generating diagnostic formulas can also be applied to this more efficient algorithm. In this paper, however, we have elected to consider the less efficient but simpler algorithm in order to highlight the principles underlying the generation of diagnostic information.

### The Postprocessing Step

After the *PREORDER* terminates the second step computes distinguishing formulas using the tuples contained in the stack. The procedure relies on the fact that if $p \not\sqsubsetsim q$ then the tuple containing $\langle p, q\rangle$ is in the stack with some additional information which explains why this is so. So a postprocessing step can process these tuples to compute formulas. The pseudocode for this step is contained in Figure 5. The intuition is as follows.

1: If the tuple is of type $[1, \langle p, q\rangle a, p']$ then either $q$ does not have an $a$-derivative and $p$ has one, or $a$-derivative $p'$ of $p$ is not related by $\sqsubsetsim$ to any of the $a$-derivatives of $q$. In the former case the formula $\langle a\rangle tt$ is satisfied by $p$ but is not satisfied by $q$. In the latter case one has to recursively build formulas

```
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
          [1, ⟨p,q⟩, a, p′]    : R = { s′ | q -a-> s′ };
                                 foreach s′ ∈ R do
                                     Φ′ = DFG(⟨p′, s′⟩, stack);
                                     Γ = Γ ∪ {Φ′};
                                 end do
                                 if Γ = ∅ then return ⟨a⟩tt;
                                 else return ⟨a⟩(∧Γ);
          [2, ⟨p,q⟩, a]        : return [a]↓tt;
          [3, ⟨p,q⟩, a, q′]    : R = { s′ | p -a-> s′ };
                                 foreach s′ ∈ R do
                                     Φ′ = DFG(⟨s′, q′⟩, stack);
                                     Γ = Γ ∪ {Φ′};
                                 end do
                                 if Γ = ∅ then return [a]↓ff
                                 else return [a]↓(∨Γ);
        endcase
    endelse
end DFG
```

**Fig. 5.** Code for computing distinguishing formulas.

that distinguish $p'$ from each $a$-derivative of $q$ and take the conjunction of them (call it $\Phi$). Then $p$ satisfies the formula $\langle a\rangle\Phi$ but $q$ can not.

2: If the tuple is of type $[2, \langle p,q\rangle, a]$ then $p$ is $a$-convergent and $q$ is $a$-divergent. The formula generated then is $[a]_{\downarrow}tt$. $q$ can not satisfy this because it diverges on action $a$.

3: If the tuple is of type $[3, \langle p,q\rangle, a, q']$ then the situation is the dual of the one shown in 1. Either $p$ does not have an $a$-derivative, or one $a$-derivative $q'$ of $q$ is not related by $\precsim$ to any of the $a$-derivatives of $p$. Note that both $p$ and $q$ are $a$-convergent. In the former case this implies that $p$ satisfies the formula $[a]_{\downarrow}ff$, since it does not have any $a$-derivative, and $q$ can not satisfy it because it does have an $a$-derivative $q'$ which can not satisfy $ff$. In the latter case, for each $p_i$ such that $p \xrightarrow{a} p_i$, we may recursively build formulas $\Phi_i$ such that $p_i \models \Phi_i$ but $q' \not\models \Phi_i$. This implies that $p \models [a]_{\downarrow}(\vee\Phi_i)$ but $q \not\models [a]_{\downarrow}(\vee\Phi_i)$.

**Theorem 10.** *Let $\mathcal{P}$ and $\mathcal{Q}$ be two processes. If $\mathcal{P} \not\precsim \mathcal{Q}$ then DFG will return a formula $\Phi$ such that $\mathcal{P} \models \Phi$ but $\mathcal{Q} \not\models \Phi$.*

The formulas generated by $DFG$ may be represented (as sets of propositional equations so that common subformulas may be shared) in space proportional to $|P|\times|Q|\times max(|P|,|Q|)$. This is due to the fact that the number of total recursive calls made by the algorithm is bounded by $|P|\times|Q|$ and each distinguishing formula is of the form $\langle a\rangle\Phi$ or $[a]_{\downarrow}\Phi$ where $\Phi$ contains at most $max(|P|,|Q|)$ conjuncts or disjuncts. If the procedure is modified such a way that we save and use this information appropriately then we have the following bound on the amount of computation needed to compute these equations.

**Theorem 11.** *An equational representation of* $DFG(\langle p,q\rangle, stack)$ *may be calculated in* $O(|P|\times|Q|\times max(|P|,|Q|))$ *time, based on the information in the stack.*

## 5 An Example

Figure 6 gives two labeled transition systems $\mathcal{P}$ and $\mathcal{Q}$ for which $\mathcal{P}\not\lesssim_W\mathcal{Q}$, where $\lesssim_W$ is discussed in Section 2.2. In order to show that $\mathcal{P}\not\lesssim_W\mathcal{Q}$ and generate the corresponding diagnostic formula we apply the method outlined in Section 2.1. First *elts* $\langle P, Act-\{\tau\}\cup\{\epsilon\},\Rightarrow,\Uparrow\rangle$ and $\langle Q, Act-\{\tau\}\cup\{\epsilon\},\Rightarrow,\Uparrow\rangle$ are constructed for $\mathcal{P}$ and $\mathcal{Q}$ and the prebisimulation algorithm is then applied to these. As a result, in the diagnostic information to be generated, $\Downarrow$ will be used in the formulas involving $[]_{\downarrow}$. Note that for all $p$ in $\mathcal{P}$, $p\Downarrow i$ and $p\Downarrow o$, while in $\mathcal{Q}$, $q_0\Downarrow i$, $q_0\Downarrow o$ and $q_1\Downarrow o$, but $q_1\Uparrow i$ , $q_2\Uparrow i$ and $q_2\Uparrow o$ . The stack is initially empty. After the first iteration the following pairs will be pushed into the stack since they are found to be $\not\lesssim_1$:

| |
|---|
| $[1,\langle p_2,q_2\rangle,o,p_1]$ |
| $[2,\langle p_2,q_1\rangle,i]$ |
| $[1,\langle p_2,q_0\rangle,o,p_1]$ |
| $[1,\langle p_1,q_2\rangle,i,p_2]$ |
| $[2,\langle p_1,q_1\rangle,i]$ |
| $[1,\langle p_1,q_0\rangle,o,p_0]$ |
| $[1,\langle p_0,q_2\rangle,i,p_1]$ |
| $[3,\langle p_0,q_1\rangle,o,q_0]$ |

After the second iteration $[1,\langle p_0,q_0\rangle,i,p_1]$ is pushed into stack because $p_0\overset{i}{\Rightarrow}p_1$, $q_0\overset{i}{\Rightarrow}q_1$ and $p_1\not\lesssim_1 q_1$. Since $\langle p_0,q_0\rangle$ is in the stack process $\mathcal{P}$ is not smaller than process $\mathcal{Q}$. In order to build the formula $DFG(\langle p_0,q_0\rangle, stack)$, the algorithm first locates the tuple which has the pair $\langle p_0,q_0\rangle$. In this case the tuple $[1,\langle p_0,q_0\rangle,i,p_1]$ is in the stack and the action causing this element to be pushed into the stack is $i$. The formula that will be returned, then, will be

$$\langle i\rangle(DFG(\langle p_1,q_1\rangle, stack))$$

The algorithm then locates the tuple containing $\langle p_1,q_1\rangle$; it is $[2,\langle p_1,q_1\rangle,i]$. The tag 2 indicates the reason why this tuple is in the stack. $q_1$ is divergent on action

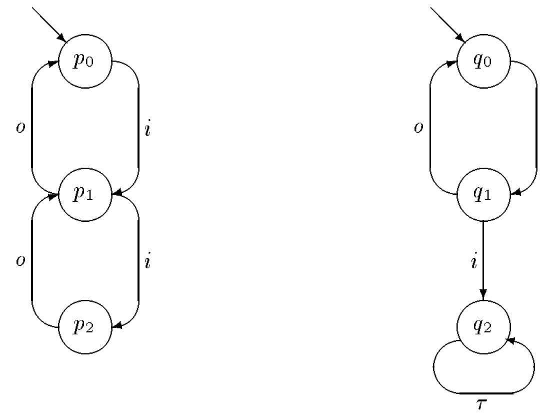


**Fig. 6.** Processes $\mathcal{P}$ and $\mathcal{Q}$ such that $\mathcal{P} \not\lesssim_W \mathcal{Q}$.

$i$ whereas $p_1$ is convergent. So the formula returned for $DFG(\langle p_1, q_1\rangle, stack)$ is

$$[i]_{\Downarrow} tt$$

This means the formula distinguishing $p_0$ from $q_0$ is

$$\langle i\rangle [i]_{\Downarrow} tt$$

Process $\mathcal{P}$ may engage in an $i$-transition from its start state $p_0$ and after that it may evolve to a convergent state on *all* $i$-transitions. (Note that there is only one such transition in this example.) However process $\mathcal{Q}$ may evolve to an $i$-divergent state on action $i$ from its start state $q_0$.

## 6 Applications of Prebisimulation Preorder

Various other preorders in the literature can be seen to be instances of the prebisimulation preorder applied to special kinds of *elts*. In this section we investigate the form that diagnostic information takes when our methodology is applied to the computation of the preorders. Let $\mathcal{P} = (\langle P, Act, \rightarrow, \uparrow\rangle, p_0)$ and $\mathcal{Q} = (\langle Q, Act, \rightarrow, \uparrow\rangle, q_0)$ be two processes.

### Bisimulation Equivalence

Bisimulation equivalence is defined in terms of bisimulations [18].

**Definition 12 (Bisimulation Equivalence).** A relation $R \subseteq P \times Q$ is a bisimulation between $P$ and $Q$ if $pRq$ implies the following.

1. $p \xrightarrow{a} p' \Rightarrow \exists q'.\ q \xrightarrow{a} q' \wedge p'Rq'$.
2. $q \xrightarrow{a} q' \Rightarrow \exists p'.\ p \xrightarrow{a} p' \wedge p'Rq'$.

$\mathcal{P} \sim \mathcal{Q}$ is defined to hold if there is a bisimulation $R$ with $p_0 R q_0$.

If $\uparrow = \emptyset$ (i.e. all states are completely defined on all actions.) then prebisimulation preorder becomes bisimulation equivalence [18]. The reason is that, in Definition 3, Condition 2 becomes $q \xrightarrow{a} q' \Rightarrow \exists p'.\, p \xrightarrow{a} p' \wedge p' R q'$ when $\uparrow$ is $\emptyset$.

Since all states are defined on all actions the convergence requirement in satisfying $[]_{\downarrow}$ is fulfilled trivially. Thus the modal operators $[]_{\downarrow}$ and $\langle\rangle$ become duals of each other. The other consequence of $\uparrow = \emptyset$ is that the formulas do not contain the subformula $[a]_{\downarrow} tt$. This is due to the fact that the tuple $[2, \langle p, q\rangle, a]$ which causes this subformula to be generated can never occur in the stack.

### Simulation Preorder

The simulation preorder, $\lesssim$, is defined in terms of simulations.

**Definition 13 (Simulation Preorder).** Let $\mathcal{P}$ and $\mathcal{Q}$ be processes. A relation $R \subseteq P \times Q$ is a simulation between $P$ and $Q$ if $pRq$ implies the following.

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

$\mathcal{P} \lesssim \mathcal{Q}$ holds if there is a simulation $R$ with $p_0 R q_0$.

If $\uparrow = P \times Act$ (so every state is considered to be *underdefined* or *incomplete* on every action) then $\sqsubseteq\!\!\!\sim$ coincides with the *simulation preorder*. The reason for this is that when $\uparrow = P \times Act$, Condition 2 in Definition 3 of prebisimulation preorder is always true, since it is never the case that $p \downarrow a$ holds.

As an immediate result of the elimination of checking Condition 2 in Definition 3, the diagnostic formulas we generate in this setting do not contain disjunctions or the modal operator $[]_{\downarrow}$, since $DFG$ generates a formula containing $[]_{\downarrow}$ or $\vee$'s only when the second condition in the definition of $\sqsubseteq\!\!\!\sim$ is violated. Then the syntax of the generated formulas turns out to be $\Phi ::= tt \mid \Phi \wedge \Phi \mid \langle a \rangle \Phi$.

### Trace Containment

The trace containment preorder is defined in terms of the sequences of actions a process may perform.

**Definition 14.**

- Let $s = a_1 \ldots a_n$ be in $Act^*$. Then $p \xrightarrow{s}$ holds if there exists $p_1, \ldots, p_n$ such that $p \xrightarrow{a_1} p_1 \ldots \xrightarrow{a_n} p_n$.
- $\mathcal{P} \sqsubseteq\!\!\!\sim_{trace} \mathcal{Q}$ if for any $s \in Act^*$ such that $p \xrightarrow{s}$, $q \xrightarrow{s}$.

If $\mathcal{P}$ and $\mathcal{Q}$ are deterministic and $\uparrow = P \times Act$ then the relation computed by the preorder checking algorithm turns out to be *trace* or *language* containment. $\mathcal{P} \sqsubseteq\!\!\!\sim_{trace} \mathcal{Q}$ in this sense exactly when $\mathcal{Q}$ is capable of engaging any sequence of actions that $\mathcal{P}$ is capable of. In this setting when $\mathcal{P} \not\sqsubseteq\!\!\!\sim_{trace} \mathcal{Q}$ then $\exists s \in Act^*$ such that $p_0 \xrightarrow{s}$ but $q_0 \not\xrightarrow{s}$.

The formulas generated by $DFG$ have a very simple form when the processes are deterministic and $\upharpoonright = P \times Act$, and it is straightforward to exhibit a sequence $s$ that $\mathcal{P}$ is capable of but $\mathcal{Q}$ is not when $\mathcal{P} \not\sqsubseteq_{trace} \mathcal{Q}$ based on this formula. In particular the formulas do not contain

- disjunctions,
- conjunctions,
- $[]_{\downarrow}$,
- $ff$.

The reason why the formulas have this special form is as follows. Since the processes are deterministic, every state in $\mathcal{P}$ and $\mathcal{Q}$ has at most one $a$-derivative for any $a$, and at each stage in $DFG$ we therefore need to distinguish an $a$-derivative of $p \in P$ from at most one $a$-derivative of $q \in Q$. This removes the necessity to use conjunction or disjunction. The formulas do not contain $[]_{\downarrow}$ because Condition 2 of Definition 3 can not be violated.

More precisely the syntax of the formulas is $\Phi ::= tt \mid \langle a \rangle \Phi$. Since the formulas have the following simple form

$$\langle a_1 \rangle ... \langle a_n \rangle tt$$

they can easily be transformed into distinguishing sequences of the form

$$a_1 ... a_n.$$

**Testing Preorders and Equivalences**

If the compatibility relation $\Pi$ mentioned in Section 2.2 is initialized correctly and the processes are transformed appropriately then testing preorders and equivalences can also be computed by the preorder checking algorithm. However, this procedure is beyond the scope of this paper, and interested reader should refer [6, 7] for a detailed account.

## 7 Concluding Remarks

One approach to verifying processes involves the use of behavioral preorder; an implementation may be deemed to satisfy a specification if it is greater than the specification. In this paper we have presented a method for computing diagnostic information for a particular preorder, the *prebisimulation preorder*. This preorder has a logical characterization in terms of a variant of Hennessy-Milner logic that enables us to generate formulas explaining why one process is not greater than the other. The generation of the formulas relies on a postprocessing step that is invoked on a stack-based representation of the information computed by the preorder information. As future work we plan to incorporate this distinguishing formula capability into the Concurrency Workbench [8], a tool for the analysis of finite-state systems. We would also like to investigate applying our techniques to Binary Decision Diagram-based algorithms [5] for computing preorders.

## References


1. Abramsky, S., **"Observation Equivalence as a Testing Equivalence"**, *Theoretical Computer Science,* vol. 53, (1987), 225-241.
2. Bergstra, J.A., and J.W. Klop, **"Process Algebra for Synchronous Communication"**, *Information and Control 60,* (1984), 109-137.
3. Bolognesi, T. and E. Brinksma, **"Introduction to the ISO Specification Language LOTOS"**, *Computer Networks and ISDN Systems,* vol. 14, (1987), 25-59.
4. Brookes, S.D., C.A.R. Hoare, and A.W. Roscoe, **"A Theory of Communicating Sequential Processes"**, *Journal of the ACM,* vol. 31, no. 3, (1984),560-599.
5. Burch, J.R., E.M. Clarke, K.C McMillan, D.L. Dill, L.J. Hwang. **"Symbolic Model Checking:** $10^{20}$ **States and Beyond,"** *In Proceedings LICS'90,* (1990).
6. Celikkan, U., and R. Cleaveland, **"Computing Diagnostic Tests for Incorrect Processes"** In *Proceedings of the Protocol Specification Testing and Verification, 12, 1992.*
7. 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.*
8. 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.
9. Cleaveland, R., **"On Automatically Distinguishing Inequivalent Processes"**, *In Proceedings of the Workshop on Computer-Aided Verification, 1990.*
10. Cleaveland, R., and B. Steffen, **"When is 'Partial' Adequate? A Logic Based Proof Technique Using Partial Specifications"**, *In Proceedings LICS'90,* (1990).
11. Cleaveland, R., and B. Steffen, **"Computing Behavioral Relations, Logically"**, *In Proceedings of ICALP'90,* (1991).
12. DeNicola, R., and M.C.B. Hennessy, **"Testing Equivalences for Processes"**, *Theoretical Computer Science,* vol. 24, (1984), 83-113.
13. Godskeen, J.C., K.G. Larsen, and M. Zeeberg, **"TAV – Tools for automatic verification"**, R89-19, Aalborg University, Denmark.
14. 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.
15. Hennessy, M., *Algebraic Theory of Processes,* MIT Press, Boston, 1988.
16. Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall, London, 1985.
17. Larsen, K.G., and B. Thomsen, **"Compositional Proofs by Partial Specification of Processes"**, Report R 87-20, University of Aalborg, July 1987.
18. Milner, R., *Communication and Concurrency*, Prentice Hall, 1989.
19. Stirling, C., **"Modal Logics for Communicating Systems"**, *Theoretical Computer Science,* vol. 49, (1987), 311-347.
20. Walker, D., **"Bisimulations and Divergence"**, *In Proceedings of the Third Annual Symposium on Logic in Computer Science*, (1988), 186-192.