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

November 13, 1995

Abstract

This paper describes a method for generating diagnostic information that explains why a given finite-state system fails to be greater than its specification with respect to the prebisimulation preorder. The information takes the form of a logical formula satisfied by the specification but not by the system and thus may be used by system designers for debugging purposes. Our technique relies on modifying an algorithm for computing the prebisimulation preorder so that information needed for generating these distinguishing formulas is saved appropriately. As a number of other behavioral preorders may be characterized in terms of prebisimulation preorder, our approach may be used as a basis for computing diagnostic information for these relations as well.

1 Introduction

Research in the area of process algebras has sparked interest in *behavioral relations* as bases for system verification [5, 19, 25, 26, 28]. In one approach, one uses a *preorder* to relate specifications, given as processes with certain states marked as "incomplete" (or "under construction"), and implementations, which typically take the form of "fully defined" processes; a system is deemed correct if it is larger than its specification in the preorder in the sense of providing "at least" the behavior dictated by the specification. These relations have not received as much attention in the literature as behavioral equivalences [6, 9, 12, 18, 19, 21, 22, 26], but the fact that they permit the use of partial specifications gives system implementors greater flexibility in developing correct implementations than equivalence-based approaches. This partiality can also be exploited when developing specifications for components that are to be used in particular network contexts [15, 24, 28], 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 overall system. In addition, preorders may be used as a basis for stepwise-refinement development methodologies in which a refinement step consists of replacing an incompletely defined component of a system by a more

*Research supported by NSF/DARPA grant CCR-9014775, NSF grant CCR-9120995, ONR Young Investigator Award N00014-92-J-1582, and NSF Young Investigator Award CCR-9257963.

fully specified one. At least two automated tools [14, 23] include algorithms for computing various 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* [2, 28], for finite-state processes; 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 specifications. The prebisimulation preorder 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* [18, 19]), the simulation preorder and the testing/failures preorder [13, 19]. 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, while Section 4 gives our method for generating diagnostic information. Section 5 then presents an example illustrating how the diagnostic information produced by our technique may be used to facilitate the stepwise development of systems. The next section shows how the method may be used to generate diagnostic information for preorders other than the prebisimulation preorder, and the paper concludes with a discussion of our results and directions for future research. An appendix also gives an efficient algorithm for computing the preorder and shows how the diagnostic information methodology works in conjunction with it.

2 Processes, Preorders, and Hennessy-Milner Logic

This section introduces the model of processes used in this paper and defines the prebisimulation preorder. It also gives an alternative characterization of the preorder in terms of formulas in a variant of the Hennessy-Milner logic [20] that processes satisfy.

2.1 Transition Systems

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

Definition 2.1 *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; and*
3. $\rightarrow \subseteq P \times Act \times P$ *is the transition relation.*

Intuitively, P represents the set of possible computation states the system may be in, Act contains the actions the it may execute, and $\rightarrow$ describes the state transitions that may

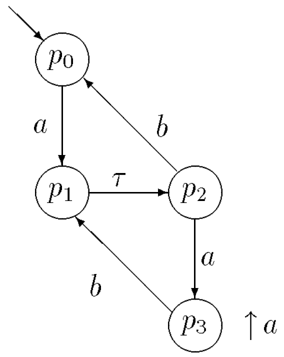

Figure 1: A Process.

result from the execution of an action in a state. We use $p \stackrel{a}{\rightarrow} p'$ in place of $(p, a, p') \in\rightarrow$ and refer to p' in this case as an *a-derivative* of p.

Extended labeled transition systems may now be given as follows.

Definition 2.2 *An extended labeled transition system (elts) is a quadruple* $\langle P, Act, \rightarrow, \uparrow\rangle$ *where* $\langle P, Act, \rightarrow\rangle$ *is a lts and* $\uparrow \subseteq P \times Act$ *is the* underdefinedness *relation.*

When $\langle p, a\rangle \in\uparrow$ holds we write $p\uparrow a$, and we use $p\downarrow a$ to abbreviate $\neg(p \uparrow a)$. Intuitively, relation $\uparrow$ is supposed to capture a notion of *underspecification.* If $p\uparrow a$ then the behavior of p on action a has not been given completely; other a-transitions from this state might be added later. Dually, when $p\downarrow a$ is true then all of p's a-transitions have been defined and p's immediate behavior is thus completely specified. 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 restrict our attention to *finite-state elts*, where $\langle P, Act, \rightarrow, \uparrow\rangle$ is finite-state if $|P| < \infty$ and $|Act| < \infty$; note that these conditions imply the finiteness of $\rightarrow$ and $\uparrow$. We also use the following notation.

- $\{p \stackrel{a}{\rightarrow} \bullet\} = \{\ p' \mid p \stackrel{a}{\rightarrow} p'\ \}$ is the set of a-derivatives of p.
- $p \stackrel{a}{\rightarrow}$ holds if $\{p \stackrel{a}{\rightarrow} \bullet\} \neq \emptyset$.
- $\{\bullet \stackrel{a}{\rightarrow} p'\} = \{\ p \mid p \stackrel{a}{\rightarrow} p'\ \}$ is the set of states having p' as an a-derivative.
- $\stackrel{a}{\rightarrow} p'$ holds if $\{\bullet \stackrel{a}{\rightarrow} p'\} \neq \emptyset$.
- $\{p \stackrel{\bullet}{\rightarrow}\} = \{\ a \mid p \stackrel{a}{\rightarrow}\ \}$ is the set of actions p is initially capable of.
- $\{\stackrel{\bullet}{\rightarrow} p'\} = \{\ a \mid \stackrel{a}{\rightarrow} p'\ \}$ is the set of actions labeling transitions leading into p'.
- $\{p\uparrow \bullet\ \} = \{\ a \mid p\uparrow a\ \}$ is the set of actions on which p diverges.

Given an *elts*, a process 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; note that state p_3 is a-divergent while all other states are convergent for all actions.

For technical reasons it is sometimes convenient to restrict processes to their reachable state space. Formally, let $M = \langle P, Act, \rightarrow, \uparrow\rangle$ be an *elts*, and let $p \in P$. Then the set

$RS(M,p)$ of states in M reachable from p is the smallest set containing p that is closed with respect to $\xrightarrow{a}$ for all $a \in Act$. Also, if $S \subseteq P$ is a set of states, then we use $M\lceil S$ to refer to the *elts* that results by restricting M to S; this is defined as

$$M\lceil S = \langle S, Act, \{\, \langle p,a,p'\rangle \in \rightarrow \mid p,p' \in S\,\}, \{\, \langle p,a\rangle \mid p \in S\,\}\rangle.$$

Then we define $R(M,p) = M\lceil RS(M,p)$, and if $\mathcal{P} = (M,p_0)$ is a process, we use $R(\mathcal{P})$ to refer to the process $(R(M,p_0),p_0)$. At an intuitive level, it is easy to see that $R(\mathcal{P})$ has exactly the same behavior as $\mathcal{P}$; in the next section, we make this intuition precise.

(Extended) labeled transition systems have been used to give semantics to a variety of different process algebras, included CSP [21], CCS [26] and LOTOS [8]. Typically, such specification formalisms include an *internal action* that represents system execution that should not be visible to a user of the system. In order to achieve a degree of abstraction from the precise amount of internal computation a process performs, a construction exists for generating an *elts* from a *lts* when the set *Act* of actions contains a such an internal action, which we henceforth refer to as τ. The construction introduces a new transition relation $\Rightarrow$ in which internal computation is "absorbed" into visible transitions and a divergence relation $\Uparrow$ reflecting the possibility of infinite internal computation. Let $\langle P, Act, \rightarrow\rangle$ be a *lts*; we may define the following.

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

Intuitively, $p_1 \overset{a}{\Rightarrow} p_2$ holds if from p_1 some internal computation may lead to a state in which an a is performed with additional internal computation then leading to p_2. If $p \Uparrow a$ is true then p may be triggered by means of an a action into an infinite internal computation. The quadruple $\langle P, (Act - \{\tau\}) \cup \{\epsilon\} \Rightarrow, \Uparrow\rangle$ now constitutes an *elts*; it is the one considered, for example, in [28].

2.2 The Prebisimulation Preorder

The prebisimulation preorder, $\sqsubseteq$, is a reflexive and transitive relation that relates processes on the basis of their behavior and relative "incompleteness" [28]. The relation is defined in terms of other relations called *prebisimulations*. Intuitively, a prebisimulation is a "matching" between states of *elts* $\langle P, Act, \rightarrow, \uparrow\rangle$ and $\langle Q, Act, \rightarrow, \uparrow\rangle$ that satisfies two 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, since as a result of "completing" p with respect to a, additional a-transitions may be added that might match q's currently unmatched a-transitions. The formal definition may be given as follows.

Definition 2.3 *Let $M = \langle P, Act, \rightarrow, \uparrow\rangle$ and $N = \langle Q, Act, \rightarrow, \uparrow\rangle$ be* elts. *Then a relation $R \subseteq P \times Q$ is a prebisimulation between M and N if pRq implies the following.*

$p \models_M tt$
$p \models_M \Phi_1 \wedge \Phi_2$ if $p \models_M \Phi_1$ and $p \models_M \Phi_2$
$p \models_M \Phi_1 \vee \Phi_2$ if $p \models_M \Phi_1$ or $p \models_M \Phi_2$
$p \models_M \langle a \rangle \Phi$ if $\exists q \in P.\ p \xrightarrow{a} q$ and $q \models_M \Phi$
$p \models_M [a]_\downarrow \Phi$ if $p \downarrow a$ and $\forall q.$ if $p \xrightarrow{a} q$ then $q \models_M \Phi$

Figure 2: Semantics of IHML.

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

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

If $\mathcal{P} = (M, p_0)$ *and* $\mathcal{Q} = (M, q_0)$ *are processes, then we say that* $\mathcal{P} \sqsubseteq\!\!\!\sim \mathcal{Q}$ *if there is a prebisimulation* R *between* M *and* N *with* $p_0 R q_0$.

It is straightforward to establish that for any *elts* M and N there is a unique largest prebisimulation between M and N and that $\precsim$ is reflexive and transitive. We sometimes abuse notation by writing $p \precsim q$ if p is a state in M, q is a state in N, and there is a prebisimulation between M and N that relates p to q. It is also the case that $\mathcal{P} \precsim R(\mathcal{P})$ and $R(\mathcal{P}) \precsim \mathcal{P}$, and thus that these two processes have the "same behavior."

There is a close connection between $\precsim$ and the relation $\precsim_W$ studied by Walker [28]. Suppose $\mathcal{P}, \mathcal{Q}$ are of the form (M, p_0) and (N, q_0), where M and N are *elts*. Let $\mathcal{P}', \mathcal{Q}'$ be $(M', p_0), (N', q_0)$, where M' and N' are the *elts* resulting from applying the construction in Section 2.1 to M and N. Then $\mathcal{P} \precsim_W \mathcal{Q}$ iff $\mathcal{P}' \precsim \mathcal{Q}'$.

2.3 Intuitionistic Hennessy-Milner Logic

The prebisimulation preorder has a logical characterization in terms of Intuitionistic Hennessy-Milner Logic (IHML) [27]. 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, $\models_M$, that indicates when states in *elts* M satisfy formulas. Formally, let $M = \langle P, Act, \rightarrow, \uparrow \rangle$; then $\models_M$ is the smallest relation satisfying the rules given in Figure 2, where $p \in P$. Intuitively, every state p satisfies the formula tt, or "true". State p satisfies $\Phi_1 \wedge \Phi_2$ if it satisfies both the Φ_i and $\Phi_1 \vee \Phi_2$ if it satisfies one of them. The constructs $\langle a \rangle$ and $[a]_\downarrow$ are modalities; p satisfies $\langle a \rangle \Phi$ if one of its a-derivatives satisfies Φ, while it satisfies $[a]_\downarrow \Phi$ if it is a-convergent and all of its a-derivatives satisfy Φ.

IHML differs from classical Hennessy-Milner Logic (HML) [20] in the semantics of the $[a]_\downarrow$ operator. HML includes no sensitivity to divergence, and hence $\langle a \rangle$ and $[a]_\downarrow$ are duals. This is not the case in IHML; if $p \models_M [a]_\downarrow \Phi$ then p must be a-convergent in addition to having all its a-derivatives satisfying Φ. This requirement reflects the intuition that if p is not a-convergent, then not all of p's a-derivatives are known, and hence statements about all of p's a-derivatives cannot be made.

$\models$ may be extended to processes in the obvious way: if $\mathcal{P} = (M, p_0)$ is a process then $\mathcal{P} \models \Phi$ if $p_0 \models_M \Phi$.

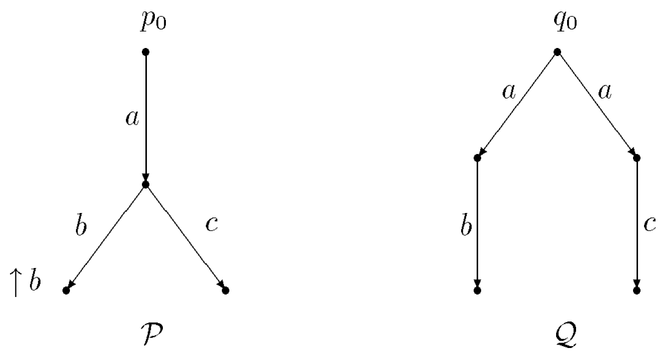

Figure 3: Example Processes.

The logical characterization of $\sqsubseteq\!\!\!\!\!\sim$ states that $\mathcal{P} \precsim \mathcal{Q}$ if and only if every formula satisfied by $\mathcal{P}$ is also satisfied by $\mathcal{Q}$ [27]. Formally let

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

be the set of IHML formulas that a process $\mathcal{P}$ satisfies. Also define *elts* M to be *image-finite* if for every state p and action a of M, $|\{p \xrightarrow{a} \bullet\}| < \infty$, and process $\mathcal{P} = (M, p_0)$ to be image-finite if *elts* $R(M, p_0)$ is; note that if $\mathcal{P}$ is finite-state, then it is also trivially image-finite. Stirling [27] has shown the following.

Theorem 2.4 *Let $\mathcal{P}$ and $\mathcal{Q}$ be image-finite. Then $H(\mathcal{P}) \subseteq H(\mathcal{Q})$ iff $\mathcal{P} \precsim \mathcal{Q}$.*

As a result of this theorem, it follows that when $\mathcal{P}$ and $\mathcal{Q}$ are finite-state and $\mathcal{P} \not\precsim \mathcal{Q}$, there must be a formula Φ such that $\mathcal{P} \models \Phi$ 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 2.5 *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. Then IHML formula Φ* 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 3. The formula $\Phi = \langle a\rangle(\langle b\rangle tt \wedge \langle c\rangle tt)$ distinguishes $\mathcal{P}$ from $\mathcal{Q}$, as p_0 has an a-derivative that is capable of actions b and c, while q_0 cannot. Thus $\mathcal{P} \not\precsim \mathcal{Q}$. In addition, the formula $\langle a\rangle\langle b\rangle[b]_{\downarrow} tt$ distinguishes $\mathcal{Q}$ from $\mathcal{P}$ since process $\mathcal{P}$ is *b-divergent* after performing actions a and b while $\mathcal{Q}$ is not. Hence $\mathcal{Q} \not\precsim \mathcal{P}$.

3 Computing the Preorder

The prebisimulation preorder has an iterative characterization that may be used as a basis for algorithms for finite-state processes. This alternative definition is as follows.

Definition 3.1 *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 $\precsim_k \subseteq P \times Q$ can be defined as follows:*

1. *$\precsim_0 = P \times Q$.*

$k := 1$;
$\precsim_0 := P \times Q$;
$\precsim_1 := \mathcal{F}(\precsim_0)$;
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*

Figure 4: Algorithm $PREORDER$.

2. $\precsim_{k+1} = \mathcal{F}(\precsim_k)$ *where*

$$\begin{aligned}\mathcal{F}(R) \quad = \{\langle p,q\rangle \mid \quad & (p \xrightarrow{a} p' \Rightarrow \exists q'.\, q \xrightarrow{a} q' \wedge p'Rq') \ \wedge \\ & (p\!\downarrow\! a \Rightarrow [q\!\downarrow\! a \wedge (q \xrightarrow{a} q' \Rightarrow \exists p'.\, p \xrightarrow{a} p' \wedge p'Rq')])\}.\end{aligned}$$

Note that R is a prebisimulation if and only if $R \subseteq \mathcal{F}(R)$. We now have the following [27].

Theorem 3.2 *Let $\mathcal{P}$ and $\mathcal{Q}$ be image-finite processes with start states p_0 and q_0 respectively. Then $\mathcal{P} \precsim \mathcal{Q}$ iff $\forall\, k \quad p_0 \precsim_k q_0$.*

On the basis of this characterization $\precsim$ can be computed by repeatedly applying $\mathcal{F}$ to $\precsim_0$ until $\precsim_{k+1} = \precsim_k$. This suggests the algorithm $PREORDER$ in Figure 4 for computing $\precsim$.[1]

4 Generating Diagnostic Information

In this section we present our technique for computing distinguishing formulas for processes that are not related by the prebisimulation preorder. The procedure consists of two phases. The first uses a slightly modified preorder-checking algorithm to gather information about states that have been determined to be unrelated, while the second transforms this information into distinguishing formulas, given a pair of states that have been found to be unrelated. We describe each of these steps in turn.

4.1 Gathering Diagnostic Information

From the definition of $\mathcal{F}$ and $\precsim_k$ it follows that if $p \not\precsim_k q$ then either $p \not\precsim_{k-1} q$ or at least one of the three conditions below must hold.

1. $\exists p'.\, p \xrightarrow{a} p'$ and $\forall q'(q \xrightarrow{a} q'$ implies $p' \not\precsim_{k-1} q')$.

2. $p\!\downarrow\! a$ but $q\!\uparrow\! a$.

3. $p\!\downarrow\! a$ and $\exists q'.\, q \xrightarrow{a} q'$ and $\forall p'(p \xrightarrow{a} p'$ implies $p' \not\precsim_{k-1} q')$.

[1]A more efficient algorithm is given in an appendix.

```
PREORDER(𝒫 : (⟨P, Act, →⟩, p₀); 𝒬 : (⟨Q, Act, →⟩, q₀)) → (boolean, stack);
  k := 1;
  stack := ∅;
  ⊑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 and 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;
```

Figure 5: Modified $PREORDER$.

These observations suggest that when p and q are found not to be related, the relevant information for explaining why this is the case depends on which of the above conditions hold for the pair. If the first condition holds, then the information can take the form of an action a and a state p' unrelated to any a-derivative of q; if the second is true, then appropriate information would be an action a such that $p\downarrow a$ but $q\uparrow a$; and if the third condition obtains, then the information could take the form of an action a with $p\downarrow a$ and state q' that is not related to any a-derivative of p. We therefore alter algorithm $PREORDER$ to record this information by pushing an appropriate *tuple* onto a stack when a pair of states are found not to be related. The tuples have the following formats.

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

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

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

The first item in a tuple is a tag indicating which of the three conditions above is violated, while the component $\langle p, q\rangle$ records the pair of states that violated this condition. The interpretation of the other parts of a tuple depend on the tag value. When the tag is 1, a is an action and p' is a state with the property that $p \xrightarrow{a} p'$ and p' is not less than any a-derivative of q. When the tag is 2, a represents an action for which $p\downarrow a$ holds but $q\uparrow a$.

Finally, a tag value of 3 indicates that a is an action such that $p\downarrow a$ while q' is a state such that $q \xrightarrow{a} q'$ and no a-derivative of p is less than q'. As an example, if $p \not\precsim q$ by virtue of the fact $p\downarrow a$ and $q\uparrow a$ then $[2, \langle p, q\rangle, a]$ could be pushed onto the stack, and if $p \xrightarrow{a} p'$ but $q \stackrel{a}{\not\rightarrow}$ then $[1, \langle p, q\rangle, a, p']$ could be. Figure 5 contains a version of $PREORDER$ in which these tuples are generated and pushed during the computation of the $\precsim_k$. We assume that semantics of the **case** statement ensures that if a pair satisfies more than one condition, then exactly one of the cases is selected nondeterministically and executed.

We now remark on some properties that hold of tuples that the modified $PREORDER$ pushes onto its stack. We first introduce the following definitions.

Definition 4.1 *A stack is a diagnostic stack if*

- *every element has form* $[1, \langle p, q\rangle, a, p']$, $[2, \langle p, q\rangle, a]$ *or* $[3, \langle p, q\rangle, a, q']$,
- *whenever* $[1, \langle p, q\rangle, a, p']$ *is in the stack then* $p \not\precsim q$ *and* $p \xrightarrow{a} p'$,
- *whenever* $[2, \langle p, q\rangle, a]$ *is in the stack then* $p\downarrow a$ *and* $q\uparrow a$,
- *whenever* $[3, \langle p, q\rangle, a, q']$ *then* $p \not\precsim q$, $p\downarrow a$ *and* $q \xrightarrow{a} q'$.

Definition 4.2 *A diagnostic stack has the no-duplicates property if no two tuples in the stack contain the same state-pair component.*

Definition 4.3 *A diagnostic stack has the inclusion property if it satisfies the following two conditions.*

- *If* $t = [1, \langle p, q\rangle, a, p']$ *is in the stack then for every* q' *such that* $q \xrightarrow{a} q'$ *the stack contains a tuple below* t *with state component* $\langle p', q'\rangle$.
- *If* $t = [3, \langle p, q\rangle, a, q']$ *is in the stack then for every* p' *such that* $p \xrightarrow{a} p'$ *the stack contains a tuple below* t *with state component* $\langle p', q'\rangle$.

Theorem 4.4 *$PREORDER$ generates a diagnostic stack with the inclusion and no-duplicates properties.*

Proof. It is straightforward to establish that the stack produced by $PREORDER$ is a diagnostic stack. The inclusion property follows from the fact that since $\precsim_{k-1}$ is computed before $\precsim_k$, all the pairs that are $\not\precsim_{k-1}$ are pushed onto the stack before those that are $\not\precsim_k$. The semantics of the case statement ensures the stack has the no-duplicates property. $\square$

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 4.5 *Let* $|\rightarrow| = |\rightarrow_P| + |\rightarrow_Q|$ *and* $|\uparrow| = |\uparrow_P| + |\uparrow_Q|$.

1. *The time complexity of the algorithm $PREORDER$ is* $O(|P|^2 \times |Q|^2 \times (|\rightarrow| + |\uparrow|))$.
2. *The space complexity of the algorithm $PREORDER$ is* $O(|P| \times |Q| + |\rightarrow| + |\uparrow|)$.

Proof.

(1) In the worst case the **while** loop executes $|P| \times |Q|$ times (this would happen if each iteration removes exactly one pair from $\sqsubseteq\!\!\!\!\sim_k$). The time spent on computing $\mathcal{F}$ is bounded by $|P| \times |Q| \times$(time to check if one of the three conditions holds for a state pair). Checking whether or not one of the three conditions holds takes $O(|\!\rightarrow_P\!| + |\rightarrow_Q| + (|\!\uparrow_P\!| + |\!\uparrow_Q\!|))$ if the action list on which a state p diverges ($\{p \uparrow \bullet\ \}$) is sorted. Thus the total work done amounts to $O(|P|^2 \times |Q|^2 \times (|\!\rightarrow_P\!| + |\!\uparrow_P\!| + |\!\rightarrow_Q\!| + |\!\uparrow_Q\!|))$ or $O(|P|^2 \times |Q|^2 \times (|\!\rightarrow\!| + |\!\uparrow\!|))$.

(2) The amount of space needed to hold the relation $\sqsubseteq\!\!\!\!\sim$ is $O(|P| \times |Q|)$, and the maximum size of the stack is $O(|P| \times |Q|)$. The amount of space needed to hold the transitions and the divergence predicates is $O(|\!\rightarrow\!| + |\!\uparrow\!|)$. Thus total amount of space required is bounded by $O(|P| \times |Q| + |\!\rightarrow\!| + |\!\uparrow\!|)$.

□

It should be noted that more efficient algorithms are possible; the appendix, for example, presents an $O(|\mathcal{P}| \times |Q| + |P| \times |\mathcal{Q}|)$-time algorithm, where $|\mathcal{P}| = |P| + |\!\rightarrow_P\!| + |\!\uparrow_P\!|$. In this section, however, we have elected to consider the less efficient but simpler *PREORDER* in order to highlight the principles underlying the generation of diagnostic information.

4.2 Generating Formulas from Diagnostic Information

The second step for building distinguishing formulas uses the tuples contained in the diagnostic stack produced by the prebisimulation preorder checker. To construct a formula distinguishing p from q, our procedure relies on the diagnostic stack having the inclusion and no-duplicates properties in addition to containing a tuple including $\langle p, q\rangle$. The pseudocode for this step is contained in Figure 6; the intuition is as follows.

1: If the tuple containing $\langle p, q\rangle$ has form $[1, \langle p, q\rangle, a, p']$ then no a-derivative of q is larger than p' in the prebisimulation preorder. In this case, if one recursively builds formulas Φ_i that distinguish p' from each a-derivative q_i of q it then follows that p satisfies the formula $\langle a\rangle \bigwedge_i \Phi_i$ while q does not. Note that we take the empty conjunction to be the formula tt; so if q has no a-derivatives then then the formula distinguishing p from it would be $\langle a\rangle tt$.

2: If the tuple containing $\langle p, q\rangle$ is $[2, \langle p, q\rangle, a]$ then p is *a-convergent* and q is *a-divergent*. From the semantics of $[a]_\downarrow$, it follows that p satisfies $[a]_\downarrow tt$ while q cannot.

3: If the tuple containing is $[3, \langle p, q\rangle, a, q']$ then p is a-convergent and no a-derivative of p is less than q'. Thus, if we can recursively construct formulas Φ_i that distinguish p_i from q' for each a-derivative p_i of p, then it follows that p satisfies $[a]_\downarrow \bigvee_i \Phi_i$ while q cannot. Note that the empty disjunction is *ff*, and thus if p has no a-transitions then the generated formula would be $[a]_\downarrow ff$.

The next theorem establishes the correctness of DFG.

Theorem 4.6 *Let M and N be elts with states p, q, and let* stack *be a diagnostic stack with the inclusion and no-duplicates properties containing a tuple with state-pair component $\langle p, q\rangle$. Then $DFG(\langle p, q\rangle, \text{stack})$ will return a formula Φ such that $p \models_M \Phi$ but $q \not\models_N \Phi$.*

```
DFG (⟨p, q⟩, stack) → Φ;
    Locate tuple in stack such that ⟨p, q⟩ ∈ tuple
    Γ := ∅;
    case tuple of
        [1, ⟨p, q⟩, a, p′]     : R := { s′ | q -a-> s′ };
                                 foreach s′ ∈ R do begin
                                     Φ′ := DFG(⟨p′, s′⟩, stack);
                                     Γ := Γ ∪ {Φ′};
                                 end;
                                 return ⟨a⟩(∧Γ);
        [2, ⟨p, q⟩, a]         : return [a]↓tt;
        [3, ⟨p, q⟩, a, q′]     : R := { s′ | p -a-> s′ };
                                 foreach s′ ∈ R do begin
                                     Φ′ := DFG(⟨s′, q′⟩, stack);
                                     Γ := Γ ∪ {Φ′};
                                 end;
                                 return [a]↓(∨Γ);
    end;
end; {* DFG *}
```

Note: We assume that *stack* is passed as a value parameter, not a reference parameter. Thus different invocations of DFG will not affect the contents of *stack*.

Figure 6: Code for Computing Distinguishing Formulas.

Proof. Let $stack[i]$ refer to the i^{th} element from the bottom of $stack$; so $stack[1]$ represents the first element pushed onto $stack$. If $\langle p, q\rangle$ is contained in some element of $stack$, we call the *height* of $\langle p, q\rangle$ in $stack$ the (unique) i such that $stack[i]$ contains $\langle p, q\rangle$. The proof proceeds by strong induction on the height of $\langle p, q\rangle$ in $stack$; so suppose the height of $\langle p, q\rangle$ in $stack$ is k. The induction hypothesis states that for any $\langle p',q'\rangle$ with height $i < k$, $DFG(\langle p', q'\rangle, stack)$ distinguishes p' from q'. To show that $DFG(\langle p, q\rangle, stack)$ generates a formula distinguishing p from q, we proceed with a case analysis on the format of $stack[k]$.

1. $stack[k] = [1, \langle p, q\rangle, a, p']$. Since $stack$ is a diagnostic stack with the inclusion property, it follows that if q has a-derivatives $q_1, \ldots, q_m$ then $p' \not\sqsubseteq q_i$ for each i, that each $\langle p', q_i\rangle$ appears in some tuple in $stack$, and that the height of each $\langle p', q_i\rangle$ is less than k. Thus, on the basis of the induction hypothesis it follows that for each i, $\Phi_i = DFG(\langle p', q_i\rangle, stack)$ distinguishes p' from q_i. From the semantics of IHML, we may then conclude that $\Phi = \langle a\rangle(\Phi_1 \wedge \cdots \wedge \Phi_m)$ is satisfied by p but not by q.

2. $stack[k] = [2, \langle p, q\rangle, a]$. It follows that $p\!\downarrow\! a$ while $q\!\uparrow\! a$. The semantics of IHML then guarantees that p satisfies $[a]_\downarrow tt$ while q does not.

3. $stack[k] = [3, \langle p, q\rangle, a, q']$. By virtue of the fact that $stack$ is diagnostic and has the inclusion property, we know that $p\!\downarrow\! a$ and that if p has a-derivatives $p_1, \ldots, p_n$ then there exist pairs $\langle p_i, q'\rangle$ in $stack$ whose heights are less than k. From the induction hypothesis, then, it follows that $\Phi_i = DFG(\langle p_i, q'\rangle, stack)$ distinguishes p_i from q'.

The semantics of IHML allow us therefore to conclude that $\Phi = [a]_{\downarrow}(\Phi_1 \vee \cdots \vee \Phi_m)$ is satisfied by p but not by q.

$\square$

As a corollary of this theorem, it follows that if $\mathcal{P} \not\sqsubseteq \mathcal{Q}$ then the stack returned by the algorithm *PREORDER* can be used by *DFG* to generate a formula Φ such that $\mathcal{P} \models \Phi$ and $\mathcal{Q} \not\models \Phi$.

The formulas generated by *DFG* may be represented in space proportional to $|P| \times |{\rightarrow_Q}| + |Q| \times |{\rightarrow_P}|$ using equations so that common subformulas may be shared [17]. If *DFG* is modified in 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 4.7 *An equational representation of $DFG(\langle p, q\rangle, stack)$ may be calculated in $O(|P| \times |{\rightarrow_Q}| + |Q| \times |{\rightarrow_P}|)$ time.*

Proof. To compute such an equational representation, we modify *DFG* as follows. First, given a pair of states $\langle p', q'\rangle \in P \times Q$, let $X_{p',q'}$ be a unique propositional variable associated with $\langle p', q'\rangle$. The modified *DFG* works as follows. We use S to maintain a set of variables of the form $X_{p',q'}$ for which an equation needs to be generated; initially, this set contains only $X_{p,q}$. The algorithm then repeatedly removes a variable $X_{p',q'}$ from S and builds a right-hand side for the variable as follows. If *stack* contains tuple $[1, \langle p', q'\rangle, a, p'']$, then the right-hand side is $\langle a\rangle(X_{p'',q_1} \wedge \cdots \wedge X_{p'',q_n})$, where $\{q_1, \ldots, q_n\} = \{q \xrightarrow{a} \bullet\}$. Right-hand sides for other tuple types are built similarly. Then an equation containing this right-hand side, with $X_{p',q'}$ as the left-hand side, is added to the equation set under construction, and the X_{p',q_i} that have not yet had right-hand sides built for them are added to S. This process is repeated until S is empty. Note that this algorithm is not recursive.

To calculate the complexity of this procedure, first note that the total number of equations to be computed is bounded by $|P| \times |Q|$, since in the worst case each state in P needs to be distinguished from every state in Q. In addition, the right hand side of an equation is of the form $\langle a\rangle\Phi$ or $[a]_{\downarrow}\Phi$ where the size of of Φ is bounded by the number of a-transitions emanating from the relevant state. Using hashing techniques, we may locate the address of tuples in *stack* containing a given pair $\langle p, q\rangle$ in $O(1)$ time. Then the time bound is

$$\begin{aligned}
\sum_{p\in P}\sum_{q\in Q} max(\sum_{p'\in\{p\xrightarrow{a}\bullet\}} O(1), \sum_{q'\in\{q\xrightarrow{a}\bullet\}} O(1)) &\leq \sum_{p\in P}\sum_{q\in Q}(\sum_{p'\in\{p\xrightarrow{a}\bullet\}} O(1) + \sum_{q'\in\{q\xrightarrow{a}\bullet\}} O(1)) \\
&\leq \sum_{p\in P}\sum_{q\in Q}(|\{p \xrightarrow{\bullet}\}| + |\{q \xrightarrow{\bullet}\}|) \\
&\leq \sum_{p\in P}(|Q||\{p \xrightarrow{\bullet}\}| + |{\rightarrow_Q}|) \\
&\leq O(|Q| \times |{\rightarrow_P}| + |P| \times |{\rightarrow_Q}|)
\end{aligned}$$

$\square$

5 Stepwise Refinement

In this section we illustrate the utility of our diagnostic information-generation method by showing how the formulas computed may be used to debug designs during the stepwise development of a simple communications protocol. We begin by showing how the prebisimulation preorder may be used in support of stepwise refinement; we then introduce a simple design language, CCS [26], that we then use to develop our protocol.

5.1 Prebisimulations and Stepwise Refinement

In a stepwise-refinement system development strategy [3, 4], a user "refines" high-level system designs by successively providing lower- and lower-level details until one arrives at an implementation that is correct with respect to a given specification. The motivation for such development strategies is that they provide designers with a framework for building structured designs in which components have clean interfaces with one another while behaving in a manner consistent with the system's specification. Specific refinement strategies would thus typically be concerned with the following issues.

1. When is design $\mathcal{Q}$ a valid refinement of design $\mathcal{P}$?
2. When is design $\mathcal{P}$ correct with respect to specification $\mathcal{S}$?
3. Can design $\mathcal{P}$ be refined to one that is correct for specification $\mathcal{S}$?

The first point concerns what is "allowed"; when is one design an "elaboration" of another? The second issue may be seen as a "termination condition"; when is the designer done? Finally, the last question concerns whether or not the designer is "on the right track"; if a design cannot be refined into a correct implementation then it contains an error, in which case the designer should review his previous refinements in order to correct the design.

Using $\sqsubset\!\!\!\sim$, one can answer the first two of these questions in a relatively straightforward fashion. $\mathcal{Q}$ may be seen to be a valid refinement of $\mathcal{P}$ if $\mathcal{P} \sqsubset\!\!\!\sim \mathcal{Q}$; this implies that $\mathcal{Q}$ "fills in" some of the undefined aspects of $\mathcal{P}$'s behavior. In a similar vein, we would deem $\mathcal{P}$ to be correct with respect to $\mathcal{S}$ if $\mathcal{S} \sqsubset\!\!\!\sim \mathcal{P}$.

Regarding the third question, given a design $\mathcal{P}$ and a specification $\mathcal{S}$ we would ideally like a criterion that, if true, would tell us that $\mathcal{P}$ can be refined into an implementation of $\mathcal{S}$, and, if false, would indicate that no possible refinement of $\mathcal{P}$ can implement $\mathcal{S}$. In general, answering this question would depend on the specific refinement methodology used, since different methodologies would allow different refinements to be made. However, the following "approximate" condition can be applied to any $\sqsubset\!\!\!\sim$-based methodology. We deem a design $\mathcal{P}$ to be free of design errors with respect to $\mathcal{S}$ if either $\mathcal{S} \sqsubset\!\!\!\sim \mathcal{P}$ or $\mathcal{P} \sqsubset\!\!\!\sim \mathcal{S}$. Since $\sqsubset\!\!\!\sim$ is transitive, it follows that if $\mathcal{S} \sqsubset\!\!\!\sim \mathcal{P}$ then any refinement $\mathcal{Q}$ of $\mathcal{P}$ also satisfies $\mathcal{S} \sqsubset\!\!\!\sim \mathcal{Q}$. On the other hand, if $P \sqsubset\!\!\!\sim S$, then there certainly exists a refinement of $\mathcal{P}$ (i.e. $\mathcal{S}$ itself) that is correct with respect to $\mathcal{S}$.

It should be noted that this criterion is not in general "complete"; that is, there exist $\mathcal{S}$ and $\mathcal{P}$ such that $\mathcal{S} \not\sqsubset\!\!\!\sim \mathcal{P}$ and $\mathcal{P} \not\sqsubset\!\!\!\sim \mathcal{S}$ and yet for which one may find a $\mathcal{Q}$ such that $\mathcal{P} \sqsubset\!\!\!\sim \mathcal{Q}$ (i.e. $\mathcal{Q}$ is a refinement of $\mathcal{P}$) and $\mathcal{S} \sqsubset\!\!\!\sim \mathcal{Q}$ (i.e. $\mathcal{Q}$ is correct with respect to $\mathcal{S}$). However, for a large class of specifications, the criterion is complete, as we show momentarily. It is also the case that one might in any event find such a design $\mathcal{P}$ to be problematic for another reason. Consider the following observation, where $\mathcal{P} \sqsubset \mathcal{Q}$ if $\mathcal{P} \sqsubset\!\!\!\sim \mathcal{Q}$ and $\mathcal{Q} \not\sqsubset\!\!\!\sim \mathcal{P}$.

Lemma 5.1 *Suppose that $\mathcal{P}$ and $\mathcal{S}$ are such that $\mathcal{P} \not\sqsubset\!\!\!\sim \mathcal{S}$ and suppose further that there is a $\mathcal{Q}$ such that $\mathcal{P} \sqsubset\!\!\!\sim \mathcal{Q}$ and $\mathcal{S} \sqsubset\!\!\!\sim \mathcal{Q}$. Then $\mathcal{S} \sqsubset \mathcal{Q}$.*

Proof. Straightforward. □

Thus, if $\mathcal{P}$ and $\mathcal{S}$ fail our criterion and yet there exists a refinement $\mathcal{Q}$ of $\mathcal{P}$ that implements $\mathcal{S}$, then it follows that $\mathcal{S}$ must be *strictly* less than $\mathcal{Q}$. Retaining $\mathcal{P}$ as a basis for further refinement in this instance serves to restrict the set of possible implementations of $\mathcal{S}$ that are also refinements of $\mathcal{P}$; in particular, if $\mathcal{Q}$ is such that $S \precsim Q \sqsubset P$, then $\mathcal{Q}$ would never be uncovered as a possible implementation if $\mathcal{P}$ were retained in the "design path."

We now characterize a class of specifications for which our criterion is complete.

Definition 5.2 *Elts* $M = \langle P, Act, \rightarrow, \uparrow \rangle$ *is* completely specified *if* $\uparrow = \emptyset$. *Process* $\mathcal{P} = (M, p_0)$ *is completely specified if elts* $R(M, p_0)$ *is.*

That is, M is completely specified if every state is completely defined for each action, while process $\mathcal{P}$ is completely specified if each state reachable from the start state is defined on every action. The next lemma now states that completely specified processes are maximal elements with respect to $\precsim$.

Lemma 5.3 *Suppose that process* $\mathcal{P}$ *is completely specified and that* $\mathcal{Q}$ *is such that* $\mathcal{P} \precsim \mathcal{Q}$. *Then* $\mathcal{Q}$ *is completely specified and* $\mathcal{Q} \precsim \mathcal{P}$.

Proof. Since $\mathcal{P}$ is completely specified and $\mathcal{P} \precsim \mathcal{Q}$, it follows that for every reachable state q in $\mathcal{Q}$ there is a reachable state p in $\mathcal{P}$ such that $p \precsim q$. Since it is the case that $p \downarrow a$ for every reachable p in $\mathcal{P}$ and $a \in Act$, it must therefore be the case that $q \downarrow a$ for every a. Thus $\mathcal{Q}$ is completely specified. That $\mathcal{Q} \precsim \mathcal{P}$ follows from the fact that $\precsim$ is an equivalence on completely specified processes [27]. □

The following theorem is an immediate consequence of Lemmas 5.1 and 5.3; it essentially states that if specification $\mathcal{S}$ is completely specified then our criterion regarding a potential design $\mathcal{P}$ is complete.

Theorem 5.4 *Suppose* $\mathcal{S}$ *is completely specified, and suppose that* $\mathcal{P}$ *is such that* $\mathcal{S} \not\precsim \mathcal{P}$ *and* $\mathcal{P} \not\precsim \mathcal{S}$. *Then there exists no* $\mathcal{Q}$ *with* $\mathcal{S} \precsim \mathcal{Q}$ *and* $\mathcal{P} \precsim \mathcal{Q}$.

Proof. If such a $\mathcal{Q}$ were to exist, then by Lemma 5.1 it would follow that $\mathcal{S} \sqsubset \mathcal{Q}$. However, since $\mathcal{S}$ is completely specified Lemma 5.3 indicates that it is maximal with respect to $\precsim$. Thus no such $\mathcal{Q}$ can exist. □

Later in this section we show how the diagnostic information generated during the checking of $\precsim$ may be used to debug incorrect refinement steps and to determine whether a design needs more refinement in order to meet a specification. Before doing this, we first introduce the notation we use to represent our designs.

5.2 CCS and Stepwise Refinement

In the example to follow we use a variant of Milner's CCS [26] to represent designs and specifications. This section provides a brief overview of some of the features of this language; the interested reader is referred to [26] for a more detailed account.

In CCS processes are built up from a set of *atomic actions* using a collection of process constructors. To define the former, let Λ be a set of communication channels. Then an atomic action may either be an input on $\lambda \in \Lambda$ (written as λ), an output on channel λ (written as $\overline{\lambda}$) or an internal computation step τ, where $\tau \notin \Lambda$. Let Act to denote the set of

$$
\begin{array}{rcl}
\bot \xrightarrow{\tau} \bot & & \\
a.p \xrightarrow{a} p & & \\
p \xrightarrow{a} p', X = p & \Rightarrow & X \xrightarrow{a} p' \\
p \xrightarrow{a} p', a \neq \tau & \Rightarrow & p [\!] q \xrightarrow{a} p' \\
q \xrightarrow{a} q', a \neq \tau & \Rightarrow & p [\!] q \xrightarrow{a} q' \\
p \xrightarrow{\tau} p' & \Rightarrow & p [\!] q \xrightarrow{\tau} p' [\!] q \\
q \xrightarrow{\tau} q' & \Rightarrow & p [\!] q \xrightarrow{\tau} p [\!] q' \\
p \xrightarrow{a} p' & \Rightarrow & p|q \xrightarrow{a} p'|q \\
q \xrightarrow{a} q' & \Rightarrow & p|q \xrightarrow{a} p|q' \\
p \xrightarrow{a} p', q \xrightarrow{\overline{a}} q' & \Rightarrow & p|q \xrightarrow{\tau} p'|q' \\
p \xrightarrow{a} p', \pi(a) \notin L & \Rightarrow & p \backslash L \xrightarrow{a} p' \backslash L
\end{array}
$$

Figure 7: CCS Semantics.

all CCS actions; we use $a, b, \ldots$ to range over Act. Also, if $a = \overline{\lambda}$ then we define $\overline{a} = \lambda$, and if $a = \lambda$ or $a = \overline{\lambda}$ then we define $\pi(a) = \lambda$ to be the "port component" of a.

The syntax of the variant of CCS we use may now be given by the following grammar. Let Var be a countable set of *process names* ranged over by $X, Y, \ldots$, and let $L \subseteq \Lambda$.

$$p \;::=\; \bot \;\mid\; a.p \;\mid\; X \;\mid\; p [\!] p \;\mid\; p|p \;\mid\; p \backslash L$$

Note that we have excluded some traditional CCS operators (e.g. *nil* and relabeling) because we do not need them for the example to follow. We have also replaced the traditional CCS choice construct, $+$, by a true external choice construct $[\!]$; we remark more on this later. Let $Proc$ represent the set of terms generated by the grammar.

Each of the CCS operators has the following informal behavior. $\bot$ represents a completely undefined process, while $a.p$ may engage in action a and behave like p. The operator $[\!]$ represents *external choice*; process $p [\!] q$ may behave like p or q, with the initial actions of each being offered to the environment. Notice that in contrast with the traditional choice operator of CCS, internal computation steps do not resolve $[\!]$. $p|q$ represents the parallel composition of p and q, with parallel execution modeled via interleaving. In addition, in $p|q$ p and q may synchronize on *complementary actions* (input and outputs on the same port), with a τ resulting. Finally, ports in $L \subseteq \Lambda$ may be declared *local* to p using the restriction operator $\backslash L$. Process names may be assigned meaning using equations; thus, the process $X = a.X$ defines a process X that is capable of an infinite sequence of a actions.

The formal semantics of CCS takes the form of a collection of rules that inductively define the transition relation $\rightarrow$. Intuitively, $p \xrightarrow{a} p'$ holds when p may engage in action a and behave like p' afterwards. The rules may be found in Figure 7.

Since $\langle Proc, Act, \rightarrow \rangle$ constitutes a *lts*, we may use the construction given in Section 2.1 to generate an *elts*

$$\langle Proc, (Act - \{\tau\}) \cup \{\epsilon\}, \Rightarrow, \Uparrow \rangle$$

that abstracts away from the internal computation performed by CCS processes. As remarked in Section 2.2, $\sqsubseteq$ over this *elts* coincides with the preorder $\sqsubseteq_W$ used by Walker [28];

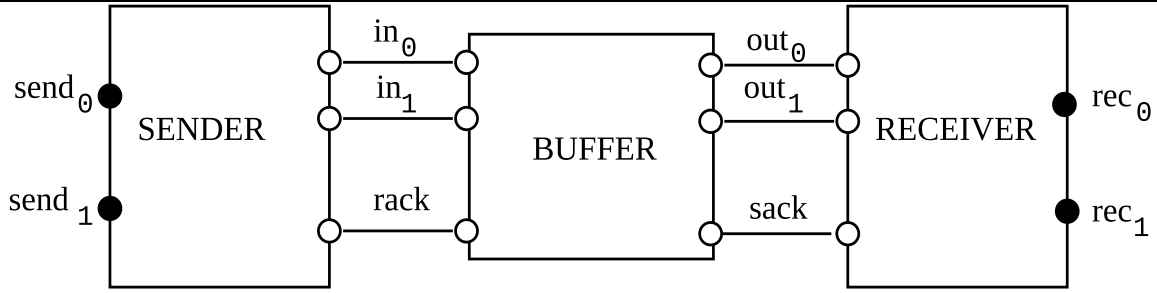

Figure 8: Architecture of the Implementation.

thus, in the remainder of this section, we write $\sqsubseteq_W$ to remind the reader that this is the preorder we use in our example. Note that since $\perp\Uparrow a$ for all $a \in (Act - \{\tau\}) \cup \{\epsilon\}$ and $\perp \stackrel{a}{\not\Rightarrow}$ for any a, it follows that $\perp \sqsubseteq_W p$ for any p. In what follows, if p is a CCS term then we write p to stand for the process whose *elts* component is $\langle Proc, (Act - \{\tau\}) \cup \{\epsilon\}, \Rightarrow, \Uparrow\rangle$ and whose start state is p.

Using the results in [28], it is straightforward to establish that $\sqsubseteq_W$ is a *precongruence* for our variant of CCS.[2] This means that $\sqsubseteq_W$ is substitutive: if $C[\ ]$ is a context (i.e. a CCS process with a "hole") and $p \sqsubseteq_W q$, then $C[p] \sqsubseteq_W C[q]$, where $C[p]$ represents $C[\ ]$ with p substituted for the hole. Note that $\parallel$ does not have a preemptive power unlike $+$. In the example we are about to consider we use a very simple notion of stepwise refinement in which instances of $\perp$ in a design are replaced by more fully specified processes as a means of developing a more refined design. Because $\perp \sqsubseteq_W p$ and $\sqsubseteq_W$ is precongruence for any p, such a strategy guarantees that the new design is a refinement of the old one.

5.3 Designing a protocol

The example we consider is a simple data-transfer protocol between two protocol entities, a sender and a receiver, connected by a medium. The sender is responsible for sending single bits to the receiver using the medium, which conveys the data from sender to receiver and acknowledgments from receiver to sender.

The service specification of this protocol requires that each bit sent by the sender be received by the end user before another bit is sent. The definition of the service specification and the medium is given below.

$$\begin{aligned} \mathcal{SPEC} &= send_0.\overline{rec_0}.\mathcal{SPEC} \parallel send_1.\overline{rec_1}.\mathcal{SPEC} \\ \mathcal{B} &= in_0.\overline{out_0}.\mathcal{B} \parallel in_1.\overline{out_1}.\mathcal{B} \parallel sack.\overline{rack}.\mathcal{B} \end{aligned}$$

Since CCS provides no direct support for data values, we instead encode the values in port names. Thus $send_0$ represents a port on which value 0 may be exchanged. The over-all structure of the protocol is illustrated in Figure 8. In order to ensure that only the sender and receiver have access to the medium, we render this architecture in CCS as

$$\mathcal{IMP} = (\mathcal{S} \mid \mathcal{B} \mid \mathcal{R})\backslash\{in_0, in_1, out_0, out_1, sack, rack\}.$$

We would now like to develop implementations of $\mathcal{S}$ and $\mathcal{R}$ that ensure that the protocol meets its service specification. Initially, we start with the following "loosest possible"

[2]In particular, since $\parallel$ is not resolved by internal transitions (unlike $+$), $\sqsubseteq_W$ is preserved by $\parallel$.

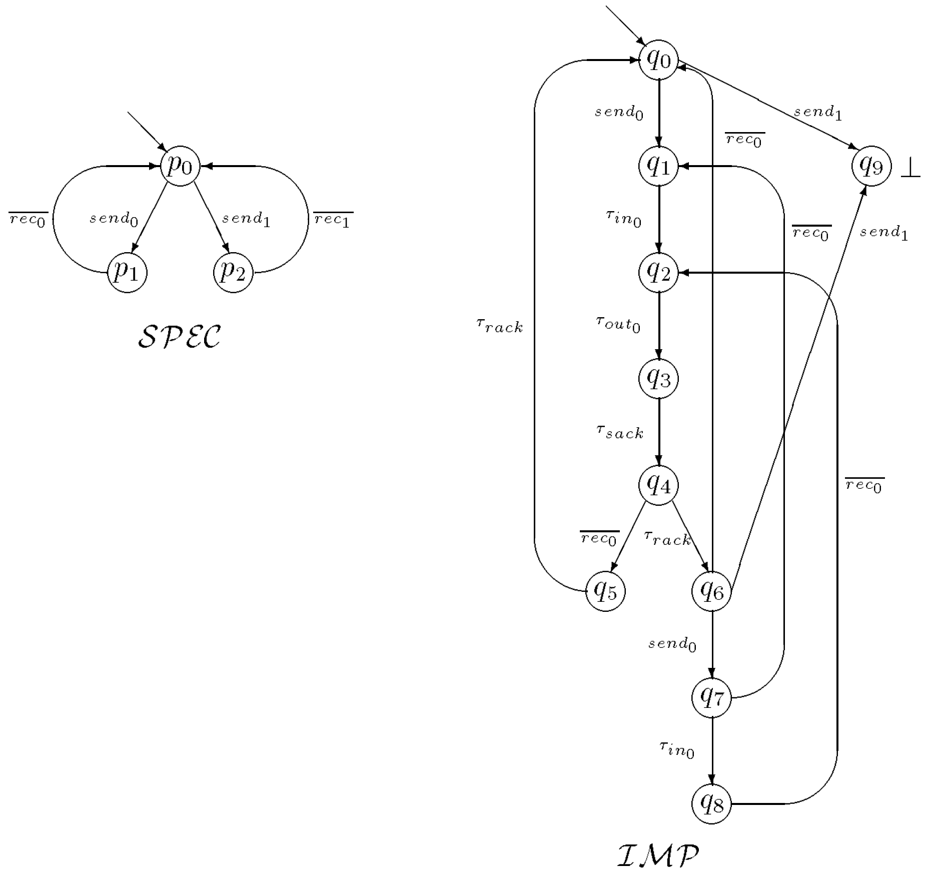

Figure 9: Processes $\mathcal{SPEC}$ and $\mathcal{IMP}$ such that $\mathcal{SPEC} \not\precsim_W \mathcal{IMP}$ and $\mathcal{IMP} \not\precsim_W \mathcal{SPEC}$.

specifications for the processes.

$$\begin{aligned} \mathcal{S} &= \perp \\ \mathcal{R} &= \perp \end{aligned}$$

Note that $(\mathcal{S} \mid \mathcal{B} \mid \mathcal{R})\backslash\{in_0, in_1, out_0, out_1, sack, rack\} \precsim_W \mathcal{SPEC}$ so this design is a reasonable starting point. We might then refine the definition of $\mathcal{S}$ and $\mathcal{R}$ by replacing instances of $\perp$ with more refined processes to arrive at the following.

$$\begin{aligned} \mathcal{S} &= send_0.\overline{in_0}.rack.\mathcal{S} \parallel send_1.\perp \\ \mathcal{R} &= out_0.\overline{sack}.\overline{rec_0}.\mathcal{R} \parallel out_1.\perp \end{aligned}$$

In this refined design $\mathcal{S}$ uses $\mathcal{B}$ to send "0" bits to $\mathcal{R}$ and awaits an acknowledgment; its behavior on sending a "1" bit has been left undefined at this point. $\mathcal{R}$ waits for the appearance of a "0" bit from the buffer and then sends an acknowledgment. Its behavior is undefined when it receives a "1" bit from the buffer. The corresponding transition graphs are shown in Figure 9. τ_{in_0} represents an internal event caused by the synchronization of in_0; it should be noted that the subscript is included for explanatory purposes only.

At this point, although our design is clearly incomplete, we wish to determine whether our design can be refined into something that is correct for $\mathcal{SPEC}$. Accordingly, we use $PREORDER$ to determine if $\mathcal{SPEC} \precsim_W \mathcal{IMP}$ (in which case $\mathcal{IMP}$ needs no further refining) or $\mathcal{IMP} \precsim_W \mathcal{SPEC}$. In the former case $PREORDER$ returns *false*; $\mathcal{IMP}$ is not a

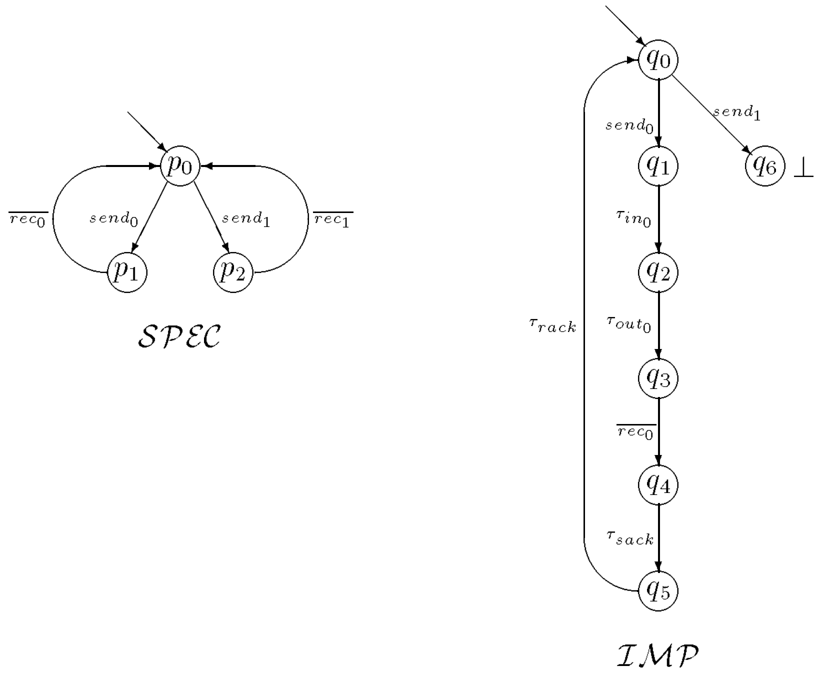

Figure 10: Corrected Sender-Receiver System.

correct implementation. This is expected, since $\mathcal{SPEC}$ has no instances of $\perp$ while $\mathcal{IMP}$ does. In the latter case, however, *false* is also returned. Since $\mathcal{SPEC}$ is completely specified, by Theorem 5.4 we conclude that no possible refinement of $\mathcal{IMP}$ can implement $\mathcal{SPEC}$. This means that $\mathcal{IMP}$ has a design error.

We now show how a diagnostic formula can be generated that points out to the source of this error. To check whether or not $\mathcal{IMP} \sqsubseteq_W \mathcal{SPEC}$, algorithm $PREORDER$ performs the following. The stack is initially empty. After the first iteration the following pair is pushed onto the stack (together with many others), since q_6 and p_1 are found to be $\not\sqsubseteq_W$.

.
$[1, \langle q_6, p_1 \rangle, send_0, q_7]$
.

After the second iteration $[1, \langle q_0, p_0 \rangle, send_0, q_6]$ is pushed onto the stack because $q_0 \overset{send_0}{\Rightarrow} q_6$, $p_0 \overset{send_0}{\Rightarrow} p_1$ and $q_6 \not\sqsubseteq_W p_1$. The stack now contains the following.

.
$[1, \langle q_0, p_0 \rangle, send_0, q_6]$
.
$[1, \langle q_6, p_1 \rangle, send_0, q_7]$
.

Since $\langle q_0, p_0 \rangle$ is in the stack $\mathcal{IMP} \not\sqsubseteq_W \mathcal{SPEC}$. In order to build a distinguishing formula for q_0 and p_0 algorithm DFG first locates the tuple which contains the pair $\langle q_0, p_0 \rangle$. In this case this tuple has the form $[1, \langle q_0, p_0 \rangle, send_0, q_6]$. The formula returned, then, is

$$\langle send_0 \rangle (DFG(\langle q_6, p_1 \rangle, stack))$$

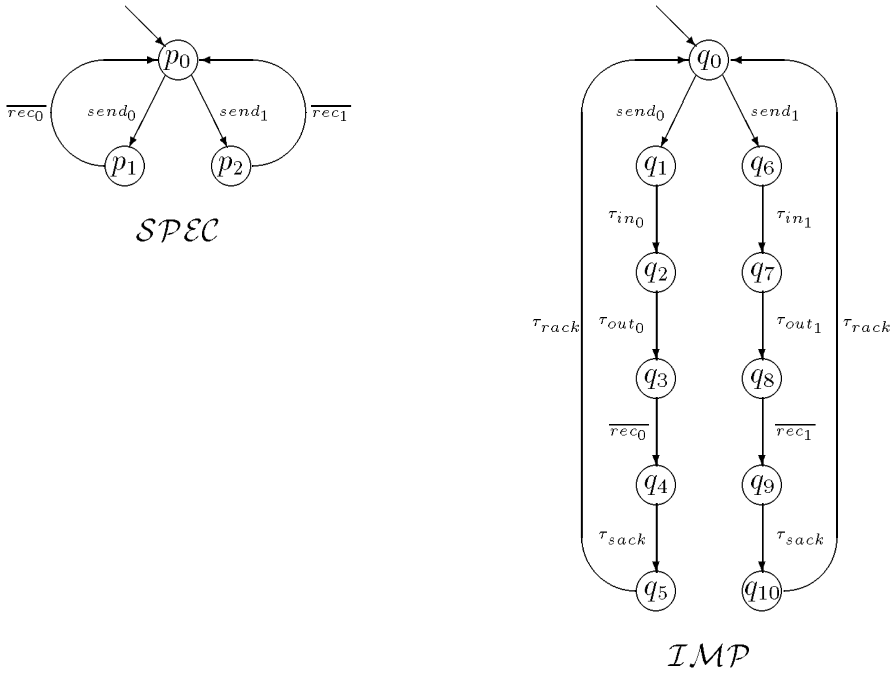

Figure 11: Final Sender-Receiver System.

The algorithm then locates the tuple containing $\langle q_6, p_1\rangle$; it is $[1, \langle q_6, p_1\rangle, send_0, q_7]$. Thus

$$DFG(\langle q_6, p_1\rangle, stack) = \langle send_0\rangle tt \quad .$$

This means the formula distinguishing q_0 from p_0 is

$$\langle send_0\rangle\langle send_0\rangle tt \quad .$$

This formula tells us that process $\mathcal{IMP}$ may engage in two successive $send_0$-transitions from its start state. In contrast, after a $send_0$ process $\mathcal{SPEC}$ can only evolve to a state in which the action $\overline{rec_0}$ is possible.

One possible remedy for this design error is to make the $\mathcal{R}$ send an $\overline{sack}$ after it delivers the "0" bit to its end user. This modification is shown below.

$$\mathcal{R} \quad = \quad out_0.\overline{rec_0}.\overline{sack}.\mathcal{R} \parallel out_1. \perp$$

Figure 10 shows the resulting transition graphs. The prebisimulation preorder algorithm returns *true* when it is executed to check if $\mathcal{IMP} \sqsubseteq\!\!\!\!\sim_W \mathcal{SPEC}$, meaning that $\mathcal{IMP}$ may be further refined into a design that is correct with respect to $\mathcal{SPEC}$. To see if further refinement is really needed, we also check whether $\mathcal{SPEC} \sqsubseteq\!\!\!\!\sim_W IMP$. The algorithm returns *false*; in this case, the diagnostic formula that is generated enables us to determine where further refinement is necessary. This formula is

$$[send_1]_{\Downarrow} tt,$$

which $\mathcal{IMP}$ does not satisfy this since $q_0 \stackrel{send_1}{\Rightarrow} q_6$ and q_6 is an incompletely specified state. By suitably adapting the implementation developed for sending 0 we arrive at the following complete specification.

$$\mathcal{S} \quad = \quad send_0.\overline{in_0}.rack.\mathcal{S} \parallel send_1.\overline{in_1}.rack.\mathcal{S}$$
$$\mathcal{R} \quad = \quad out_0.\overline{rec_0}.\overline{sack}.\mathcal{R} \parallel out_1.\overline{rec_1}.\overline{sack}.\mathcal{R}$$

The corresponding transition graphs are shown in Figure 11. Given $\mathcal{SPEC}$ and $\mathcal{IMP}$ as arguments, $PREORDER$ now returns *true*; therefore design $\mathcal{IMP}$ is correct with respect to its specification.

6 Applications of the Prebisimulation Preorder

Other preorders in the literature can be seen to be instances of the prebisimulation preorder applied to special kinds of *elts*. When DFG is used in conjunction with a prebisimulation preorder algorithm to compute these other relations, then, the distinguishing formulas produced will be restricted in form, depending on the characteristics of the *elts* being analyzed. In this section we investigate the form that diagnostic information takes when our methodology is applied to the computation of several different preorders. In what follows, let $\mathcal{P} = (\langle P, Act, \rightarrow, \uparrow\rangle, p_0)$ and $\mathcal{Q} = (\langle Q, Act, \rightarrow, \uparrow\rangle, q_0)$ be processes.

6.1 Bisimulation Equivalence

Bisimulation equivalence is defined in terms of bisimulations [26].

Definition 6.1 (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. $q \xrightarrow{a} q' \Rightarrow \exists p'.\ p \xrightarrow{a} p' \wedge p'Rq'$.

$\mathcal{P} \sim \mathcal{Q}$ holds if there is a bisimulation R between P and Q with p_0Rq_0.

If $\mathcal{P}$ and $\mathcal{Q}$ are completely specified then the prebisimulation preorder coincides with bisimulation equivalence [26]. The reason is that, in Definition 2.3, Condition 2 reduces to $q \xrightarrow{a} q' \Rightarrow \exists p'.\ p \xrightarrow{a} p' \wedge p'Rq'$ when $\uparrow = \emptyset$.

Since all states are defined on all actions the convergence requirement for formulas of the form $[a]_{\downarrow}\Phi$ 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 DFG to generate this subformula, can never occur in a stack since it is impossible for $q\uparrow a$ to hold.

6.2 Simulation Preorder

The simulation preorder, $\lesssim$, is defined in terms of simulations [20].

Definition 6.2 (Simulation Preorder) *A relation $R \subseteq P \times Q$ is a simulation between P and Q if pRq implies the following.*

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

$\mathcal{P} \lesssim \mathcal{Q}$ holds if there is a simulation R with p_0Rq_0.

If $\uparrow = P \times Act$ (so every state is considered to be *underdefined* or *incomplete* on every action) then $\underset{\sim}{\sqsubset}$ coincides with the *simulation preorder*. The reason for this is that when $\uparrow = P \times Act$, Condition 2 in Definition 2.3 is vacuously satisfied, since it is never the case that $p \downarrow a$ holds. Because of this, a preorder-checking algorithm would not generate any tuples of type 2 or 3 in its diagnostic stack, and as an immediate result, the diagnostic formulas generated by DFG in this setting will not contain disjunctions or the modal operator $[]_{\downarrow}$. Thus the syntax of the formulas generated by DFG is given by $\Phi ::= tt \mid \Phi \wedge \Phi \mid \langle a \rangle \Phi$, which is well known to be sufficient to characterize logically the simulation preorder [20].

6.3 Trace Containment

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

Definition 6.3

- *Let $s = a_1 \ldots a_n$ be in Act^*. Then $p \xrightarrow{s}$ holds if there exist $p_1, \ldots, p_n$ such that $p \xrightarrow{a_1} p_1 \ldots \xrightarrow{a_n} p_n$.*
- *$\mathcal{P} \underset{\sim}{\sqsubset}_{trace} \mathcal{Q}$ if for any $s \in Act^*$ such that $p_0 \xrightarrow{s}$, $q_0 \xrightarrow{s}$.*

Thus when $\mathcal{P} \not\underset{\sim}{\sqsubset}_{trace} \mathcal{Q}$ there must be an $s \in Act^*$ such that $p_0 \xrightarrow{s}$ but $q_0 \not\xrightarrow{s}$. Thus, when $\mathcal{P} \not\underset{\sim}{\sqsubset}_{trace} \mathcal{Q}$ appropriate diagnostic information would take the form of a trace that $\mathcal{P}$ is capable of but $\mathcal{Q}$ is not.

If $\mathcal{Q}$ is deterministic and $\uparrow = P \times Act$ then the prebisimulation preorder reduces to trace containment. In addition, if $\mathcal{Q}$ is nondeterministic then it may be transformed into a trace-equivalent $\mathcal{Q}'$ that is deterministic using standard determinization techniques for finite-state machines [13]. By combining this transformation with a prebisimulation-checking algorithm, then, one may determine when arbitrary finite-state processes are related by $\underset{\sim}{\sqsubset}_{trace}$.

When $\mathcal{Q}$ is deterministic and $\uparrow = P \times Act$ the formulas generated by DFG have an especially restricted form. From the discussion of the simulation preorder, we already know that formulas would fall in the set generated by the grammar

$$\Phi ::= tt \mid \Phi \wedge \Phi \mid \langle a \rangle \Phi$$

since $\uparrow = P \times Act$. Moreover, since $\mathcal{Q}$ is deterministic every state in $\mathcal{Q}$ 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 implies that when $\mathcal{Q}$ is deterministic, DFG generates no instance of $\wedge$. Thus the formulas returned by DFG have the form

$$\langle a_1 \rangle \cdots \langle a_n \rangle tt$$

From the semantics of IHML, it follows that a system satisfies such a formula exactly when it may perform the sequence

$$a_1 \ldots a_n \ .$$

Thus DFG may also be used to generate trace information when a prebisimulation checker is used to test processes for trace containment.

6.4 Testing Preorders

Other preorders may be seen as instances of slight alterations of the prebisimulation preorder for *elts*. Let Π be a preorder between P and Q; intuitively, Π may be thought of as reflecting an ordering on information stored in states. Then a Π-prebisimulation is a relation R such that $R \subseteq \Pi$ and R is a prebisimulation, and $\mathcal{P} \sqsubseteq_{\Pi} \mathcal{Q}$ if there exists a Π-prebisimulation relating the start states of $\mathcal{P}$ and $\mathcal{Q}$.

Testing preorders [13] turn out to be instances of the Π-prebisimulation preorder if the processes being checked are transformed appropriately. Typically, these transformations involve *determinization* of the processes as in the previous subsection; however the determinization procedure is a variant of the standard one for finite-state machines in that information is assigned to states in the determinized process. In general each state y in the "transformed" process is labeled with the initial moves of each state in the original process corresponding to the transformed state and a flag indicating whether any of these "old states" may diverge. This information is encoded as acceptance sets [19], for which a preorder Π may then be defined to turn $\sqsubseteq_{\Pi}$ into testing preorders. The interested reader should refer to [11, 13] for a detailed account of the methodology to obtain testing preorders as the instances of prebisimulation preorder and for techniques to turn formulas generated by DFG into appropriate tests that may be used to distinguish processes.

7 Concluding Remarks

In this paper we have presented a method for computing diagnostic information for the prebisimulation preorder, a semantic relation on processes. When one finite-state process fails to be larger in the preorder than its specification, our technique efficiently generates a formula in a variant of the Hennessy-Milner Logic that is satisfied by the specification but not by the process; this formula may then be used as the basis for debugging the process. We also presented an example showing how the debugging information may be used to guide the stepwise development of a system design, and we indicated how our method may be used to compute appropriate diagnostic information for other semantic preorders as well.

Several directions for future work present themselves. We are planning on implementing a distinguishing formula generator in the Concurrency Workbench [14], a tool for the analysis of finite-state systems, as a means of providing automatically generated debugging information for system designers. It would also be interesting to study the generation of diagnostic information for preorder algorithms based on binary decision diagrams (BDDs) [10]. BDDs support the symbolic representation of state spaces of systems and consequently can succinctly encode enormous numbers of states, which may then be analyzed using BDD-specific algorithms. Our method for generating diagnostic information is not immediately applicable to these BDD algorithms, since we require the recording of information for individual pairs of states as they are found not to be related, whereas the BDD-based representations are efficient to the extent that they avoid such state-by-state analysis. However, it appears that some variant of our approach may be useful. We would also like to explore the generation of diagnostic information for on-the-fly preorder-checking algorithms, which construct the state spaces and perform the checking at the same time. As these algorithms do not require the whole state space, substantial improvements in performance (especially when a design fails to meet its specification) can result. Finally, it could be fruitful to explore further the use of semantic preorders in the context of stepwise refinement. In particular, developing more exact criteria for determining when a design may be refined into a correct implementation of a given specification and when it cannot would yield valuable methodological insights that could be exploited by system designers to detect and correct design errors quickly.

Acknowledgments. The authors would like to thank the referees for their valuable comments on earlier drafts of this paper.

References

[1] *Fifth Annual Symposium on Logic in Computer Science (LICS '90)*, Philadelphia, June 1990. IEEE Computer Society Press.

[2] S. Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53:225–241, 1987.

[3] R.J.R. Back. Refinement calculus, part 2: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93, REX Workshop, Mook, The Netherlands, June 1989. Springer-Verlag.

[4] R.J.R. Back and K.Sere. Stepwise refinement of action systems. In *Mathematics of Program Construction*, volume 575 of *Lecture Notes in Computer Science*, pages 115–138, Gronningen, The Netherlands, June 1989. Springer-Verlag.

[5] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1990.

[6] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[7] B. Bloom and R. Paige. Computing ready simulations efficiently. In *Proceedings of the North American Process Algebra Workshop*, Workshops in Computing, pages 119–134, Stony Brook, New York, August 1992. Springer-Verlag.

[8] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[9] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the Association for Computing Machinery*, 31(3):560–599, July 1984.

[10] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Fifth Annual Symposium on Logic in Computer Science (LICS '90)* [1], pages 428–439.

[11] U. Celikkan and R. Cleaveland. Computing diagnostic tests for incorrect processes. In *Proceedings of the IFIP Symposium on Protocol Specification, Testing and Verification*, pages 263–278, Lake Buena Vista, Florida, June 1992. North-Holland.

[12] R. Cleaveland. On automatically explaining bisimulation inequivalence. In E.M. Clarke and R.P. Kurshan, editors, *Computer Aided Verification (CAV '90)*, volume 531 of *Lecture Notes in Computer Science*, pages 364–372, Piscataway, NJ, June 1990. Springer-Verlag.

[13] R. Cleaveland and M.C.B. Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5:1–20, 1993.

[14] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[15] R. Cleaveland and B. Steffen. When is 'partial' adequate? A logic-based proof technique using partial specifications. In *Fifth Annual Symposium on Logic in Computer Science (LICS '90)* [1], pages 440–449.

[16] R. Cleaveland and B. Steffen. Computing behavioural relations, logically. In J. Leach Albert, B. Monien, and M. Rodríguez Artalejo, editors, *Automata, Languages and Programming (ICALP '91)*, volume 510 of *Lecture Notes in Computer Science*, pages 127–138, Madrid, July 1991. Springer-Verlag.

[17] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2:121–147, 1993.

[18] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1983.

[19] M.C.B. Hennessy. *Algebraic Theory of Processes*. MIT Press, Boston, 1988.

[20] M.C.B. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–161, January 1985.

[21] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.

[22] H. Korver. Computing distinguishing formulas for branching bisimulation. In K.G. Larsen and A. Skou, editors, *Computer Aided Verification (CAV '91)*, volume 575 of *Lecture Notes in Computer Science*, pages 13–23, Aalborg, Denmark, July 1991. Springer-Verlag.

[23] K.G. Larsen, J.C. Godskesen, and M. Zeeberg. TAV—tools for automatic verification. Technical Report R 89-19, Aalborg University, Aalborg, Denmark, 1989.

[24] K.G. Larsen and B. Thomsen. Compositional proofs by partial specification of processes. Technical Report R 87-20, Aalborg University, Aalborg, Denmark, July 1987.

[25] K.G. Larsen and B. Thomsen. A modal process logic. In *Third Annual Symposium on Logic in Compter Science (LICS '88)*, pages 203–210, Edinburgh, Scotland, July 1988. IEEE Computer Society Press.

[26] R. Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.

$R := P \times Q$;
$ToDelete := R - \mathcal{F}(R)$
while $ToDelete \neq \emptyset$ **do begin**
  $Choose\ \langle p, q\rangle \in ToDelete$;
  $R := R - \{\langle p, q\rangle\}$;
  $ToDelete := R - \mathcal{F}(R)$;
end;

Figure 12: Overview of Algorithm *EFF_PREORDER*.

[27] C. Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, 49:311–347, 1987.

[28] D.J. Walker. Bisimulation and divergence in CCS. In *Third Annual Symposium on Logic in Computer Science*, pages 186–192, Edinburgh, Scotland, July 1988. IEEE Computer Society Press.

Appendix

This appendix gives a more efficient preorder-checking algorithm than the one presented in Section 4 and shows how it may be altered to generate a stack of tuples suitable for processing by the distinguishing formula generator *DFG*. The algorithm is based on the ideas presented in [7] and [16, 17] in that it looks "backwards" along transitions when a pair is deleted from a prebisimulation under construction in order to determine which other pairs still in the relation might be affected. The work in [16, 17] uses such a technique to give an efficient model-checking algorithm for a particular temporal logic and indicates how the method might be used for prebisimulation preorder checking, while that in [7] uses similar ideas to give an efficient algorithm for the ready simulation preorder.

Recall from Definition 3.1 that a relation R is a prebisimulation if and only if $R \subseteq \mathcal{F}(R)$. Thus, if $R - \mathcal{F}(R) \neq \emptyset$ then R is not a prebisimulation. It is easy to see that the algorithm *EFF_PREORDER* in Figure 12 computes the largest prebisimulation over $P \times Q$ and may thus be used to determine if $\mathcal{P} \sqsubseteq\!\!\!\!\sim \mathcal{Q}$.

Our major task in making *EFF_PREORDER* efficient involves incrementally updating the value of $ToDelete = R - \mathcal{F}(R)$ in response to changes in R. From the definition of $\mathcal{F}$ it follows that if $\langle p, q\rangle \in R - \mathcal{F}(R)$ then one of the three conditions below must hold.

1. $\exists p'.\ p \xrightarrow{a} p'$ and $\forall q'(q \xrightarrow{a} q'$ implies $p' \not\!R q')$.
2. $p{\downarrow}a$ but $q{\uparrow}a$.
3. $p{\downarrow}a,\ q \xrightarrow{a} q'$ and $\forall p'(p \xrightarrow{a} p'$ implies $p' \not\!R q')$.

To understand how we may exploit these conditions, suppose that pRq holds and that p has an a-derivative p' such that $p'Rq'$ for only one a-derivative q' of q. Now if $\langle p', q'\rangle$ is removed from R, it follows that $\langle p, q\rangle \notin \mathcal{F}(R)$ and hence $\langle p, q\rangle \in R - \mathcal{F}(R)$. Thus $\langle p, q\rangle$ must be removed from R, as R cannot be a prebisimulation if $\langle p, q\rangle \in R$. This suggests that if we know how many a-derivatives of q that p' is related to, then when this number reaches 0 we

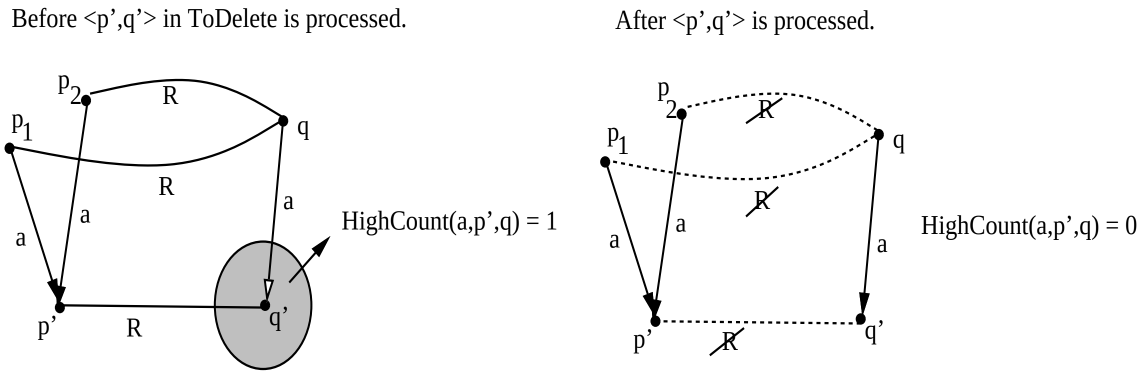

Figure 13: Illustration of the Main Loop when $HighCount$ Value is 1.

can remove pairs involving q and states (like p) having an a-transition into p'. To record this information, the elaborated version of $EFF\_PREORDER$ maintains two arrays of counters to accomplish these tasks. We describe these in turn.

- $HighCount(a, p', q)$ is the number of a-derivatives of the state q in Q that are "higher" than p'.
- $LowCount(a, p, q')$ is the number of a-derivatives of the state p in P that are "lower" than q'.

Formally they are defined as follows.

$$HighCount(a, p', q) = |\{\ q' \mid q \xrightarrow{a} q' \wedge p' R q'\ \}|$$

$$LowCount(a, p, q') = |\{\ p' \mid p \xrightarrow{a} p' \wedge p' R q'\ \}|$$

The task of implementing the algorithm $EFF\_PREORDER$ can now be divided into an initialization phase and an iteration phase. The first involves the computation of the initial value of R, $ToDelete$, $HighCount$ and $LowCount$. The initialization code is given in Figure 15. Note that $HighCount(a, p', q)$ is set to the number of a-derivatives of q since we are assuming that every state in P is related to every element in Q, and similarly for $LowCount$.

After the initialization phase, the algorithm repeatedly removes elements from $ToDelete$, deletes them from R, and incrementally updates $ToDelete = R - \mathcal{F}(R)$ using the counters $HighCount$ and $LowCount$. (As a practical matter, note that when an element is added to $ToDelete$, one may immediately remove it from R without affecting the correctness of the algorithm.) These counter values determine when a pair of states is not an element of $\mathcal{F}(R)$ and thus should be added to $ToDelete$ for eventual removal from R. The algorithm terminates when $ToDelete = \emptyset$, in which case $R = \mathcal{F}(R)$ and R is therefore the largest prebisimulation between P and Q. Figures 13 and 14 illustrate how $HighCount$ counters are used during the algorithm; $LowCount$ is employed analogously. Figure 16 contains code for the iteration phase of the algorithm.

While computing the preorder the algorithm also pushes relevant tuples onto the stack for later use by DFG. In the initialization phase, if a state p has an a-derivative whereas q does not, then a tuple of type 1 is pushed onto the stack. Similarly, if q has an a-derivative

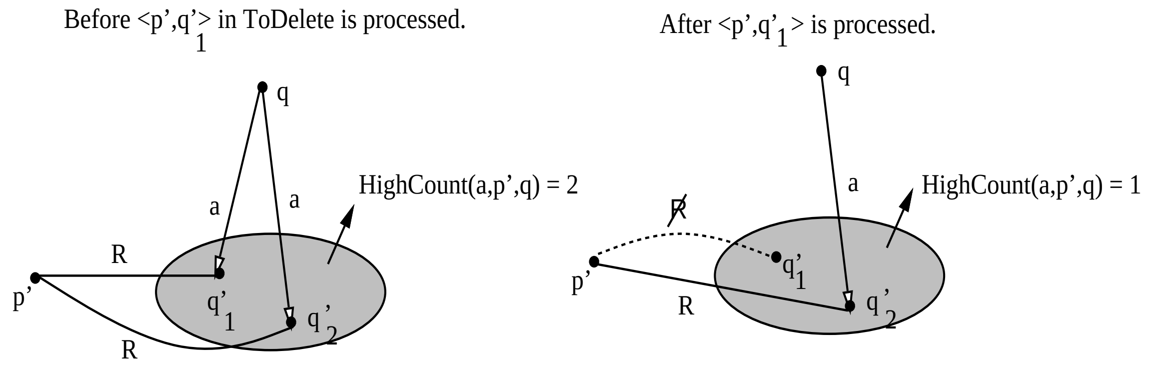

Figure 14: Illustration of the Main Loop when $HighCount$ Value is Greater Than 1.

but p does not and p is *a-convergent* then a tuple of type 3 is pushed onto the stack. Tuples of type 2 are also generated when p and q are found such that $p{\downarrow}a$ and $q{\uparrow}a$.

In the main loop of the algorithm the counters $HighCount$ and $LowCount$ are used to determine which tuples to push. A $HighCount(a, p', q)$ value of 0 means that there is no a-derivative of q related to p'; thus tuples of type 1 containing $\langle p, q\rangle$ pairs for all p such that $p \xrightarrow{a} p'$ should be stored. This situation is shown in Figure 13. A similar action is taken when a $LowCount$ value reaches 0.

The next theorem establishes that the $stack$ returned by the new algorithm $EFF\_PREORDER$ is a diagnostic stack with the inclusion and no-duplicates properties. Thus, on the basis of Theorem 4.6 DFG may be used to generate distinguishing formulas using the $stack$ produced by this algorithm.

Theorem 7.1 *$EFF\_PREORDER$ generates a diagnostic stack with the inclusion and no-duplicates properties.*

Proof. It is trivial to establish that $EFF\_PREORDER$ builds a diagnostic stack with the no-duplicates property. In the case of the inclusion property, we only prove that when $[1, \langle p, q\rangle, a, p']$ is pushed onto the stack by $EFF\_PREORDER$ then for all q' such that $q \xrightarrow{a} q'$, a tuple containing $\langle p', q'\rangle$ is already in the stack. The case for tuples of type 3 can be proved similarly and is left to the reader.

To begin with, note that tuple $[1, \langle p, q\rangle, a, p']$ may only be pushed onto the stack because $HighCount(a, p', q) = 0$. This implies that every a-derivative of q has been previously determined not to be larger than p'. Since $EFF\_PREORDER$ pushes a tuple for a pair as soon as the pair is removed from R, it follows that when $[1, \langle p, q\rangle, a, p']$ is pushed onto the stack, the stack already includes tuples containing $\langle p', q'\rangle$ for each a-derivative q' of q. □

We now remark on the time complexity of the new preorder-checking algorithm shown in Figures 15 and 16. If $\mathcal{P} = \langle\langle P, Act, \rightarrow_P, \uparrow_P\rangle, p_0\rangle$ is a process, let $|\mathcal{P}| = |P| + |\rightarrow_P| + |\uparrow_P|$. We also assume that sets are implemented using hash tables and thus that insertion, deletion and set membership may be computed in $O(1)$ time.

Theorem 7.2 *The code shown in Figure 15 takes $O(|P| \times |\mathcal{Q}| + |Q| \times |\mathcal{P}|)$ time.*

```
ToDelete := ∅;
R := P × Q;
foreach ⟨p,q⟩ ∈ { ⟨p,q⟩ | ∃a. p↓a but q↑a } do begin
   ToDelete := ToDelete ∪ {⟨p,q⟩};
   stack := PUSH([2, ⟨p,q⟩, a], stack);
end;

foreach ⟨p,q⟩ ∈ P × Q do begin
   foreach a ∈ ({p -•->} ∪ {q -•->}) do begin
      HighCount(a,p,q) = |{ q' | q -a-> q' }|;
      LowCount(a,p,q) = |{ p' | p -a-> p' }|;

{* If q does not have an a-transition, pair ⟨p,q⟩ is inserted into ToDelete. *}
      if HighCount(a,p,q) = 0 and ⟨p,q⟩ ∉ ToDelete then begin
         ToDelete := ToDelete ∪ {⟨p,q⟩};
         Choose p' from {p -a-> •};
         stack := PUSH([1, ⟨p,q⟩, a, p'], stack);
      end;

{* If p does not have an a-transition, pair ⟨p,q⟩ is inserted into ToDelete. *}
      if LowCount(a,p,q) = 0 and p↓a and ⟨p,q⟩ ∉ ToDelete then begin
         ToDelete := ToDelete ∪ {⟨p,q⟩};
         Choose q' from {q -a-> •};
         stack := PUSH([3, ⟨p,q⟩, a, q'], stack);
      end;
   end;
end;
```

Figure 15: Initialization of *ToDelete* and R.

Proof. If the action list on which a state p diverges ($\{p\uparrow\bullet\}$) is sorted then the first "`foreach`" loop takes $O(|Q| \times |\uparrow_P| + |P| \times |\uparrow_Q|)$ time. This follows from the fact that

$$\begin{aligned}\sum_{p\in P}\sum_{q\in Q} |\{p\uparrow\bullet\} \cup \{q\uparrow\bullet\}| &\leq \sum_{p\in P}\sum_{q\in Q}(|\{p\uparrow\bullet\}| + |\{q\uparrow\bullet\}|)\\ &\leq \sum_{p\in P}(|Q| \times |\{p\uparrow\bullet\}| + |\uparrow_Q|)\\ &\leq |Q| \times |\uparrow_P| + |P| \times |\uparrow_Q|.\end{aligned}$$

Similarly, the second "`foreach`" loop consumes $O(|P| \times |{\rightarrow_Q}| + |Q| \times |{\rightarrow_P}|)$ time, since

$$\sum_{p\in P}\sum_{q\in Q}\sum_{a\in(\{p\overset{\bullet}{\rightarrow}\}\cup\{q\overset{\bullet}{\rightarrow}\})} O(1) \;\leq\; \sum_{p\in P}\sum_{q\in Q} |\{p\overset{\bullet}{\rightarrow}\}| + |\{q\overset{\bullet}{\rightarrow}\}|$$

```
while ToDelete ≠ ∅ do begin
    Choose ⟨p', q'⟩ ∈ ToDelete;
    R := R − {⟨p', q'⟩};
/* First loop */
    foreach a ∈ {-•-> q'} do begin
       foreach q ∈ {• -a-> q'} do begin
          HighCount(a, p', q) := HighCount(a, p', q) − 1;
          if HighCount(a, p', q) = 0 then begin
             foreach p ∈ {• -a-> p'} do
                if ⟨p, q⟩ ∈ R and ⟨p, q⟩ ∉ ToDelete then begin
                      ToDelete := ToDelete ∪ {⟨p, q⟩};
                      stack := PUSH([1, ⟨p, q⟩, a, p'], stack);
                end;
          end;
       end;
    end;
/* Second loop */
    foreach a ∈ {-•-> p'} do begin
       foreach p ∈ {• -a-> p'} such that p↓a do begin
          LowCount(a, p, q') := LowCount(a, p, q') − 1;
          if LowCount(a, p, q') = 0 then begin
             foreach q ∈ {• -a-> q'} do
                if⟨p, q⟩ ∈ R and ⟨p, q⟩ ∉ ToDelete then begin
                      ToDelete := ToDelete ∪ {⟨p, q⟩};
                      stack := PUSH([3, ⟨p, q⟩, a, q'], stack);
                end;
          end;
       end;
    end;
    ToDelete := ToDelete − {⟨p', q'⟩};
end
```

Figure 16: Main Loop of the Algorithm $EFF\_PREORDER$.

$$
\begin{aligned}
&\leq \sum_{p\in P}\sum_{q\in Q} |\{q \xrightarrow{\bullet}\}| + \sum_{p\in P}\sum_{q\in Q} |\{p \xrightarrow{\bullet}\}| \\
&\leq \sum_{p\in P} |{\rightarrow_Q}| + \sum_{q\in Q} |{\rightarrow_P}| \\
&\leq O(|P| \times |{\rightarrow_Q}| + |Q| \times |{\rightarrow_P}|).
\end{aligned}
$$

Initialization of R takes $O(|P| \times |Q|)$ time. Thus the total amount of time spent is

$O(|P| \times |{\uparrow_Q}| + |Q| \times |{\uparrow_P}| + |P| \times |{\rightarrow_Q}| + |Q| \times |{\rightarrow_P}| + |P| \times |Q|) = O(|P| \times |\mathcal{Q}| + |Q| \times |\mathcal{P}|)$.

□

Theorem 7.3 *The code shown in Figure 16 takes $O(|P| \times |{\rightarrow_Q}| + |Q| \times |{\rightarrow_P}| + |P| \times |Q|)$ time.*

Proof. We first note that in the worst-case, the outer loop executes at most $O(|P| \times |Q|)$ times. Thus the time-complexity T of this code fragment may be characterized as follows:

$$T \;=\; T_1 + T_2 + \sum_{\langle p,q\rangle \in P \times Q} O(1)$$

where T_1 represents the total amount of time spent executing the first "`foreach`" loop over all iterations of the outermost "`while`" loop and T_2 represents the time spent in the second "`foreach`" loop. T_1 may be further decomposed into:

$$T_1 \;=\; T_i + T_r$$

where T_i represents the total amount of time spend in the innermost "`foreach` $p \in \{\bullet \xrightarrow{a} p'\}$ `do begin`" loop and T_r represents the time spent in the rest of the loop (e.g. in decrementing $HighCount(a, p, q')$ and performing the test in the `if-then` statement). From the structure of the loops, we have the following.

$$\begin{aligned}
T_r &= \sum_{\langle p',q'\rangle \in P \times Q} \; \sum_{a \in \{\xrightarrow{\bullet} q'\}} \; \sum_{q \in \{\bullet \xrightarrow{a} q'\}} O(1) \\
&= \sum_{p' \in P} \sum_{q' \in Q} \; \sum_{a \in \{\xrightarrow{\bullet} q'\}} \; \sum_{q \in \{\bullet \xrightarrow{a} q'\}} O(1) \\
&= \sum_{p' \in P} O(|{\rightarrow_Q}|) \\
&= O(|P| \times |{\rightarrow_Q}|)
\end{aligned}$$

Regarding T_i, first note that for any a, p' and q, $HighCount(a, p', q)$ has its value changed to 0 at most once. Thus the total amount of time spent in the innermost loop may be characterized as follows.

$$\begin{aligned}
T_i &= \sum_{\langle a,p',q\rangle \in Act \times P \times Q} \; \sum_{p \in \{\bullet \xrightarrow{a} p'\}} O(1) \\
&= \sum_{q \in Q} \; \sum_{\langle a,p'\rangle \in Act \times P} \; \sum_{p \in \{\bullet \xrightarrow{a} p'\}} O(1) \\
&= \sum_{q \in Q} O(|{\rightarrow_P}|) \\
&= O(|Q| \times |{\rightarrow_P}|)
\end{aligned}$$

Thus $T_1 = O(|P| \times |{\rightarrow_Q}| + |Q| \times |{\rightarrow_P}|)$. Using a similar argument, we may also infer that $T_2 = O(|P| \times |{\rightarrow_Q}| + |Q| \times |{\rightarrow_P}|)$, and the theorem then follows. □
Note that the time spent in the initialization phase of this algorithm asymptotically dominates the time devoted to the second phase. We thus have the following.

Theorem 7.4 *$EFF\_PREORDER$ takes $O(|P| \times |\mathcal{Q}| + |\mathcal{P}| \times |Q|)$ time.*

Note that DFG requires asymptotically less time than $EFF\_PREORDER$ (Theorem 4.7).