VERIFICATION OF EQUIVALENCES BETWEEN FINITE TRANSITION SYSTEMS

Theory and Applications

Tommaso Bolognesi
CNUCE/C.N.R. - Pisa

Rapporto interno C86-16

CNUCE/C.N.R.
Pisa, Dicembre 1986

VERIFICATION OF EQUIVALENCES BETWEEN FINITE TRANSITION SYSTEMS

Theory and Applications

Tommaso Bolognesi
CNUCE/C.N.R. - Pisa

Abstract

The concepts of labelled transition system, and of observation-equivalence and testing-equivalence between systems are introduced. These topics are discussed in a number of papers, sparse in the literature; they are conveniently assembled here in a unique, self-contained presentation. All proofs have been worked out in detail, and the reader is assumed *unfamiliar* with fixpoint theory. Three algorithms for verifying equivalence between finite transition systems are introduced and discussed (two for observation-equivalence and one for testing-equivalence). A proof of correctness for one of them, Sanderson's algorithm, is given (it was not found in the literature). The algorithms have been implemented in Prolog, and their application to a small example is illustrated.

Riassunto

Vengono introdotti i concetti di sistema di transizioni, e di equivalenze osservazionale e di 'testing' fra sistemi. Questi argomenti sono discussi in svariati articoli, sparsi nella letteratura; essi sono convenientemente raccolti qui in una presentazione unica ed introduttiva. Tutte le dimostrazioni sono state elaborate in dettaglio, e non si assume famigliarita' con la teoria del punto fisso. Vengono introdotti e discussi tre algoritmi per verificare la equivalenza fra sistemi di transizione finiti (due per la equivalenza osservazionale, uno per la equivalenza di 'testing'). Viene fornita una prova di correttezza di uno di essi, l'algoritmo di Sanderson (non essendo stata trovata in letteratura). Gli algoritmi sono stati implementati in Prolog, e viene illustrata la loro applicazione ad un piccolo esempio.

Key Words and Phrases
Observation equivalence, testing equivalence, verification.

This work has been partly supported by the Computer Science Dept. of Twente University of Technology, The Netherlands, in the context of a collaboration within project ESPRIT/SEDOS/C2.

Table of Contents

0. INTRODUCTION

Notions of equivalence play an important role in the formal specification of hardware and software systems. One may wish to replace a formally described subpart of a large design with an equivalent one, without affecting the overall behaviour of the system. Or a complex, formally specified implementation may be proved equivalent to a given, more abstract, specification.

Specification languages such as LOTOS [DP8807] are based on the model of **labelled transition systems**, which, in the finite case, are represented by directed graphs with labelled edges. Therefore, the design of sophisticated verification tools for the analysis of complex specifications (e.g. in LOTOS) almost inevitably requires preliminary experience on equivalence verification between finite labelled transition systems.

We discuss a few ways to define two notions of equivalence, and three algorithms for solving corresponding verification problems, which we have implemented in Prolog. Prolog is convenient for the fast development of software ("rapid prototyping"). The choice of Prolog has been determined by the need to invistigate the validity and applicabity of these algorithms by experimenting with actual prototype implementations. The counterpart of rapid prototyping is computational inefficiency, but this problem is out of the scope of this report. Only when various prototypes have been experimented with, efficient (possibly non-Prolog) implementations of the selected algorithms can be searched for.

This report is organized into two main sections. **Section 1** presents a collection of definitions and results on the notions of **'Observation equivalence'** and **'testing equivalence'** between labelled transition systems. They provide the necessary theoretical background for the various algorithms subsequently introduced. These topics are discussed in a number of papers, sparse in the literature, and are conveniently assembled here in a unique, self-contained presentation. All proofs have been worked out in detail, and the reader is assumed *unfamiliar* with fixpoint theory.
Section 2 deals with the verification of observation equivalence (algorithms "refinements" and Sanderson's "bisimulation construction") and testing equivalence (algorithm "double subset construction") between finite labelled transition systems explicitely represented as *graphs*. A proof of correctness for Sanderson's algorithm is provided. All the algorithms presented have been programmed in PROLOG, and their applications to a small example are illustrated.

1.1. Labelled transition system

Definition 1.1.1

A **labelled transition system** is a quadruple (P, Σ, T, p_0), where:

- P is a countable set of **states**;
- Σ is a countable set of **observable actions**;
- T is a $(\Sigma \cup \{\tau\})$-indexed family of **labelled transition** relations on P:

  $T = \{-\mu\rightarrow \leq P \times P \mid \mu \text{ in } \Sigma \cup \{\tau\}\}$

  where τ is the **unobservable** (or internal) **action**;
- p_0 is the **initial state**. •

An example of a labelled transition system is shown in Figure 1.1.1, where:

$P = \{p_0, p_1, p_2, p_3, p_4, p_5\}$

$\Sigma = \{a, b, c\}$

$T = \{ -a\rightarrow, -b\rightarrow, -c\rightarrow, -\tau\rightarrow\}$, with

$-a\rightarrow = \{(p_0, p_1), (p_4, p_5)\}$

$-b\rightarrow = \{(p_1, p_2)\}$

...

and p_0 is the initial state.

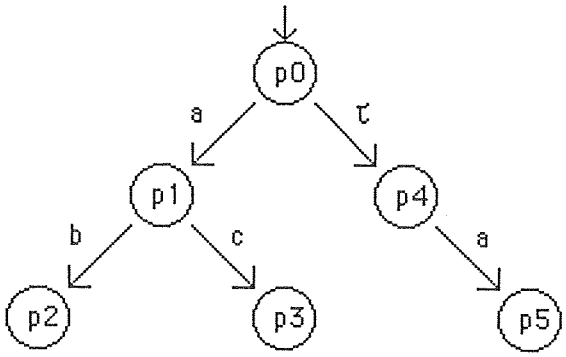

Figure 1.1.1 - A labelled transition system

We will drop the attribute "labelled" in the sequel.

Given a set A, let A^* denote the set of finite sequences of elements of A, and let ε denote the empty sequence. The following two definitions are relative to a given transition system $S(P, \Sigma, T, p_0)$.

Definition 1.1.2

The $(\Sigma \cup \{\tau\})^*$-indexed family of **action sequence** relations on P is defined as follows. If s in $(\Sigma \cup \{\tau\})^*$ is $\mu_1\mu_2...\mu_n$, with n >= 0, then:

$$-s\rightarrow = \{(p, q) | p, q \text{ in } P \text{ and there exist } p_0, p_1, ..., p_n \text{ such that}$$

$$p = p_0,\ p_0 - \mu_1 \rightarrow p_1,\ p_1 - \mu_2 \rightarrow p_2, ...,\ p_{n-1} - \mu_n \rightarrow p_n,\ p_n = q\}$$ •

Notice that in the case n=0 this definition reads: $-\varepsilon\rightarrow = \{(p, p) \mid p \text{ in } P\}$.

Definition 1.1.3

The Σ^*-indexed family of **observable action sequence** relations on P is defined as follows. If s in Σ^* is $\omega_1\ \omega_2... \omega_n$, with n >= 0, then:

$$=s\Rightarrow = \{(p, q) | p, q \text{ in } P \text{ and there exist nonnegative integers k0, k1, ..., kn such that}$$

$$p - (\tau^{k0}\ \omega_1\ \tau^{k1} ...\ \omega_n\ \tau^{kn}) \rightarrow q\}$$ •

Notice that in the case n=0 this definition reads:

$$=\varepsilon\Rightarrow = \{(p, q) \mid p, q \text{ in } P \text{ and there exists a nonegative k0 such that } p - \tau^{k0} \rightarrow q\}.$$

For example, relative to Figure 1.1.1 we have: $p_0 \text{-}\varepsilon\rightarrow p_0$, $p_0 \text{-}ab\rightarrow p_2$, $p_0 \text{-}\tau\, a\rightarrow p_5$, $p_0 =\varepsilon\Rightarrow p_0$, $p_0 =\varepsilon\Rightarrow p_4$, $p_0 =a\Rightarrow p_1$, $p_0 =a\Rightarrow p_5$.

Finally, we will write '$p =s\Rightarrow$' to mean that there exists a p' such that $p =s\Rightarrow p'$.

1.2. Function *F*

In this section we introduce a function which is essential for the definitions of observation equivalence given in Sections 1.3 and 1.4. The reader may also consult [M83] and [HM85].

Let $S(P, \Sigma, T, p_0)$ be a transition system, and let $R = 2^{PxP}$ be the set of relations over P. Function $F: R \rightarrow R$ is defined as follows:

Definition 1.2.1

If R is a relation over P, then:

$F(R) = \{(p, q) \mid p, q \text{ in } P$

and whenever $p =s\Rightarrow p'$ for some s in Σ^* and p' in P

then $q =s\Rightarrow q'$ for some q' in P, and (p', q') in R;

and whenever $q =s\Rightarrow q'$ for some s in Σ^* and q' in P

then $p =s\Rightarrow p'$ for some p' in P, and (p', q') in R $\}$ •

The four examples in Figure 1.2.1 show the effect of applying function *F* to four relations $R_1 ... R_4$ defined over the states of four transition systems. Notice that we have considered *finite* transition systems, and *equivalence* relations (i.e., relations which are reflexive, symmetric and transitive): such relations can be depicted by partitioning P into equivalence classes (first column in figure), with the understanding that p R q iff p and q are in the same class. The relations $F(R_i)$ are shown in the second column. The examples show that $F(R)$ can be strictly smaller, or equal, or strictly larger than R, and that it may also happen that neither relation include the other (third column).

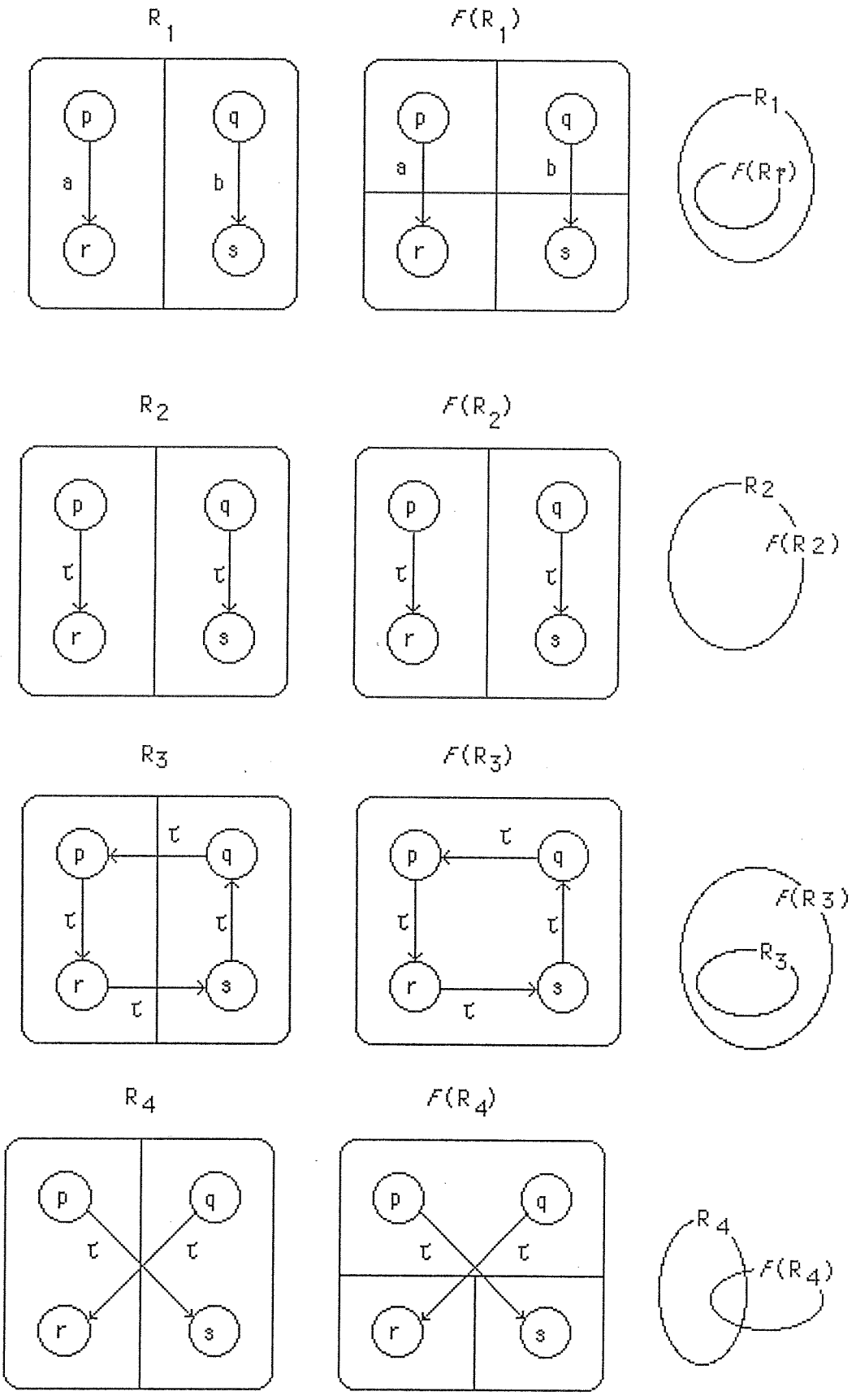

Figure 1.2.1 - Applying function *F*

Function F satisfies the following two properties.

Proposition 1.2.2

F is monotone with respect to the partial order induced by set inclusion, i.e.:

$$R_1 \leq R_2 \text{ implies } F(R_1) \leq F(R_2).$$

Proof: the proof is simply based on the definition of F. •

The reader may have noticed that, in Figure 1.2.1, we could represent $F(R_1)$... $F(R_4)$ as partitions of the state spaces. This is not accidental, as F preserves equivalence:

Proposition 1.2.3

If R is an equivalence, then $F(R)$ is an equivalence.

Proof: the reflexivity, symmetry and transitivity of $F(R)$ easily follow from these same properties of R. •

1.3. Observation equivalence via bisimulation

In this section the observation equivalence between two transition systems is defined in terms of the concept of bisimulation, which in turn is based on function F. We shall also prove that observation equivalence is the maximum fixpoint of F. The interested reader may also consult [M80], [M83] and [HM85].

Let $S(P, \Sigma, T, p_0)$ be a transition system.

Definition 1.3.1

A relation $R \leq PxP$ is a **bisimulation** if $R \leq F(R)$. •

Proposition 1.3.2

If R is a bisimulation then $F(R)$ is a bisimulation.

Proof: the proof follows from the monotonicity of F (Proposition 1.2.2). •

Definition 1.3.3

States p and q of a transition system are **observationally equivalent**, written $p \approx q$, if there exists a bisimulation R such that p R q. •

Example 1.3.4

In the transition system depicted in Figure 1.3.1 the following is a bisimulation:

$$R = \{(1, 2), (2, 3), (3, 3), (4, 5), (5, 6), (6, 6)\}$$

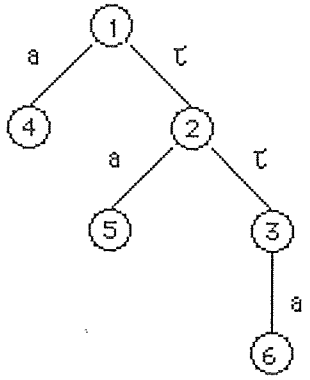

Figure 1.3.1 - A transition system

Hence, states 1 and 2 are observationally equivalent. Notice that R is neither reflexive, nor symmetric, nor transitive. •

Proposition 1.3.5

$\approx$ is an equivalence.

Proof.

$\approx$ is reflexive, because Id = PxP is a bisimulation.

$\approx$ is symmetric, because if $p \approx q$, then p R q for some bisimulation R, then q R^{-1} p, and R^{-1} is a bisimulation (which is different from saying that R is symmetric!).

$\approx$ is transitive because the composition of two bisimulations is a bisimulation. •

Definition 1.3.3 amounts to saying that $\approx$ is the union of all bisimulations:

$$\approx = \cup \{R \subseteq PxP \mid R \subseteq F(R)\}.$$

Proposition 1.3.6

$\approx$ is a bisimulation (the largest bisimulation).

Proof. If (p, q) in $\approx$, then there exists a bisimulation R such that (p, q) is in R. Since R is a

bisimulation, (p, q) is in F (R). Also, since $R \leq \approx$, then by the monotonicity of F, F (R)$\leq F(\approx)$; thus (p, q) is in $F(\approx)$, and we have proved that $\approx \leq F(\approx)$. •

Proposition 1.3.7

$\approx$ is a fixpoint of F, that is: $F(\approx) = \approx$.

Proof. By Proposition 1.3.6 we know that $\approx \leq F(\approx)$. We only need to prove $F(\approx) \leq \approx$. Since $\approx$ is a bisimulation, by Proposition 1.3.2 $F(\approx)$ is also a bisimulation, thus it is included in $\approx$, which is the union of all bisimulations. •

Proposition 1.3.8

$\approx$ is the maximum fixpoint of F.

Proof. Any fixpoint of F is a bisimulation. By Propositions 1.3.6 and 1.3.7, $\approx$ is the maximum bisimulation and is also a fixpoint. •

1.4. Observation equivalence via k-equivalences

The purpose of this section is to define Σ-observational equivalence and show that, for the special class of image-finite transition systems, it identifies with the observation equivalence defined in Section 1.3. The Σ-observational equivalence is defined via a chain of k-equivalences, in turn defined via function F. The reader may also consult [M80], where the chain of k-equivalences was first introduced, and [HM85], where the result on image-finiteness was first proved.

Let S be a transition system with state space P, and let F be the function defined in Section 1.2. We define a sequence of equivalences on P as follows.

Definition 1.4.1

$$\approx_0 = PxP$$

$$\approx_k = F(\approx_{k-1}) \text{ with k natural} > 0.$$

If $p \approx_k q$ then we say that p and q are **k-observationally equivalent.** •

Proposition 1.4.2

$\approx_k$ is an equivalence, for any natural k.

Proof: the inductive proof follows from the facts that PxP is an equivalence and that F preserves equivalence (Proposition 1.2.3). •

Proposition 1.4.3

For any natural k we have: $\approx_{k+1} \leq \approx_k$; that is, the chain $\{\approx_k\}$ is non-increasing.

Proof: the inductive proof follows from the facts that $\approx_1 \leq$ PxP and that F is monotone (Proposition 1.2.2). •

Definition 1.4.4

We say that p ad q are **ω-observationally equivalent**, written $p \approx_\omega q$, if $p \approx_k q$ for any natural k. That is:

$$\approx_\omega = \bigcap_{k=0}^{\omega} \approx_k \quad \bullet$$

Proposition 1.4.5

$\approx_\omega$ is an equivalence.

Proof. The reflexivity, symmetry and transitivity of $\approx_\omega$ directly follow from these same properties of $\approx_k$, for any k. •

Definition 1.4.6

A **relation** $\Rightarrow \leq$ PxP is **image-finite** if for each p in P the set $\{p' \mid p \Rightarrow p'\}$ is finite. •

Definition 1.4.7

A **transition system** is **image-finite** if for any s in Σ^*, relation $=s\Rightarrow$ is image-finite. •

Theorem 1.4.8

If a transition system is image-finite, then $\approx_\omega$ is identical to observation equivalence, that is $\approx_\omega$ is the maximum fixpoint of F.

Proof.

1) We show that $\approx_\omega \leq F(\approx_\omega)$. If (p, q) is in $\approx_\omega$ and $p =s\Rightarrow p'$ for some s in Σ^* and p' is in P, then for any k>=0 it must be possible to find a q'_k such that $q =s\Rightarrow q'_k$ and $p' \approx_k q'_k$. Since $=s\Rightarrow$ is image-finite, there exists a q' such that $q' = q'_k$ for infinitely many k's, and since $\{\approx_k\}$ is a

non-increasing chain of relations (Proposition 1.5.3), $p' \approx_k q'$ for *all* k's. Therefore $p' \approx_\omega q'$; and (p, q) is in $F(\approx_\omega)$.

2) We show that $F(\approx_\omega) \leq \approx_\omega$. We have that $F(\approx_\omega) \leq \approx_0$, and that $F(\approx_\omega) \leq \approx_k$ for any $k>=1$, because $\approx_\omega \leq \approx_{k-1}$ and F is monotone. Thus $F(\approx_\omega)$ is a lower bound of $\{\approx_k\}$. Since $\approx_\omega$ is by definition the largest lower bound of $\{\approx_k\}$, we conclude that $F(\approx_\omega) \leq \approx_\omega$.

3) We prove that if $R = F(R)$, then $R \leq \approx_\omega$. Induction basis: $R \leq \approx_0$. Inductive step: since F is monotone, if $R \leq \approx_k$, then $F(R)=R \leq F(\approx_k)= \approx_k$. Thus, $R \leq \approx_k$ for any k, hence R is a lower bound of $\{\approx_k\}$. $\approx_\omega$ is the largest lower bound, thus $R \leq \approx_\omega$. •

For general transition systems the two equivalences $\approx_\omega$ and $\approx$ do not coincide, and $\approx_\omega$ is not a fixpoint of F. In this case, the maximum fixpoint of F is reached by carrying the limit $\cap\{\approx_k\}$ on to transfinite ordinals.

Usually we are interested in the observation equivalence between *two* disjoint transition systems, with initial states p and q, that is, systems where the two state spaces do not intersect. This problem is solved by defining the union of the two transition systems and by verifying the equivalence between p and q in the new system.

Definition 1.4.9

Let $S_p(P, \Sigma_p, T_p, p_0)$ and $S_q(Q, \Sigma_q, T_q, q_0)$ be two transition systems. Their union is defined as:

$$(P \cup Q, \Sigma_p \cup \Sigma_q, T_p \cup T_q, x)$$

where x is an arbitrary state. •

Example 1.4.10

The two states p and q in the transition system illustrated in Figure 1.4.1 are such that $p \approx_0 q$, $p \approx_1 q$, but not$(p \approx_2 q)$, hence not$(p \approx q)$. •

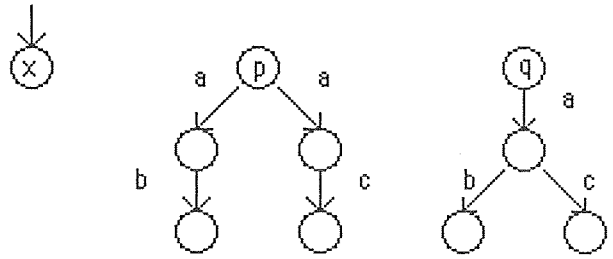

Figure 1.4.1

1.5. Observation equivalence via function *F'*

The reason for introducing function *F'* is of a practical nature: similarly to function *F*, it can be used to define observation equivalence; but unlike function *F* , it provides a basis for efficient verification algorithms.

Definition 1.5.1

Let Σ^{01} denote the set $\Sigma^0 \cup \Sigma^1$ of strings on Σ not longer than 1. If R is a relation over P, then:

$F'(R) = \{(p, q) \mid p, q \text{ in } P$

and whenever $p \Rightarrow^{s} p'$ for some s in Σ^{01} and p' in P

then $q \Rightarrow^{s} q'$ for some q' in P, and (p', q') in R;

and whenever $q \Rightarrow^{s} q'$ for some s in Σ^{01} and q' in P

then $p \Rightarrow^{s} p'$ for some p' in P, and (p', q') in R $\}$ •

Proposition 1.5.2

A relation R is a bisimulation via *F* (see Definition 1.3.1) iff it is a bisimulation via *F'*.

Proof.

We will talk about *F*- or *F'*-bisimulation.

Part a) R is an *F*-bisimulation **implies** R is an *F'*-bisimulation.

If R is an *F*-bisimulation then, when (p, q) in R, we have:

$\forall$ s in Σ^*. H(p, q, s, R),

where H(p, q, s, R) means:

if p $=s\Rightarrow$ p', for some p'

then q $=s\Rightarrow$ q' for some q' such that p' R q',

and if q $=s\Rightarrow$ q', fop some q'

then p $=s\Rightarrow$ p' for some p' such that p' R q'.

Hence, when (p, q) in R, we have, in particular:

$\forall s$ in $\Sigma^0 \cup \Sigma^1$. H(p, q, s, R),

which means that (p, q) in F'(R). Thus R $\leq F'$(R), and R is an F'-bisimulation.

Part b) R is an F'-bisimulation **implies** R is an F-bisimulation.

We must prove that R $\leq F$(R), i.e. that $\forall$(p, q) in R. $\forall$s in Σ^*. H(p, q, s, R).

This is expanded into: $\forall$(p, q) in R. $\forall n>=0$. $\forall$s in Σ^n. H(p, q, s, R),

which can be written: $\forall n>=0$. [$\forall$(p, q) in R. $\forall$s in Σ^n. H(p, q, s, R)],

or, shortly: $\forall n>=0$. K(n)

We prove K(n) by induction.

Basis. Our hypothesis R $\leq F''$(R) can be expressed as:

$\forall$(p, q) in R. $\forall$s in $\Sigma^0 \cup \Sigma^1$. H(p, q, s, R),

which means K(0) and K(1).

Step. Assume K(n), with n>=1. We want to prove K(n+1), i.e.:

$\forall$(p, q) in R. $\forall$s' in Σ^n. $\forall$a in Σ .H(p, q, s'a, R).

If (p, q) in R then, by the inductive hypothesis we have H(p, q, s', R), the first "half" of which is:

p=s'$\Rightarrow$p" for some p" implies

q $=s'\Rightarrow$ q" for some q" such that (p", q") in R.

On the other hand, if (p", q") in R, then, by hypothesis we have H(p", q", a, R), the first "half" of which is:

p" $=a\Rightarrow$ p' for some p' implies

q" $=a\Rightarrow$ q' for some q' such that (p', q') in R.

By combining the results on $=s'\Rightarrow$ and $=a\Rightarrow$ we obtain:

$p =s'a\Rightarrow p'$ for some p' implies

$q =s'a\Rightarrow q'$ for some q' such that (p', q') in R,

which is "half" of H(p, q, s'a, R). The symmetric half is proved analogously, and we conclude that H(p, q, s'a, R) holds, for any (p,q) in R and any s'a in Σ^{n+1}. •

Similarly to Definition 1.3.3 (observation equivalence $\approx$), we have

Definition 1.5.3

$p \approx' q$ iff there exists an F' - bisimulation R such that p R q. •

Proposition 1.5.4

$\approx'$ is identical to $\approx$.

Proof. It follows from Proposition 1.5.2. •

Proposition 1.5.5

$\approx'$ (thus $\approx$) is the maximum fixpoint of F'.

Proof. By arguments completely similar to those which support the analogous Proposition 1.3.8. •

Similarly to what we did in Section 1.4, we can build a non-increasing chain of relations by starting from PxP and using, now, function F':

Definition 1.5.6

$$\approx'_0 = PxP$$

$$\approx'_k = F'\ (\approx_{k-1})$$

$$\approx'_\omega = \bigcap_{k=0}^{\omega} \{F'^{\,k}(PxP)\} \quad \bullet$$

The chain $\{\approx'_k\}$ of relations over P is non-increasing. A version of Theorem 1.4.8 holds, and we have:

Theorem 1.5.7

If a transition system is image-finite, then $\approx'_\omega$ is the maximum fixpoint of F' (hence it coincides

with $\approx'$ and $\approx$).

Proof. See Theorem 1.4.8. •

Notice that ($\forall s$ in Σ^* . $=s\Rightarrow$ is image-finite) iff ($\forall s$ in $\Sigma^0 \cup \Sigma^1$. $=s\Rightarrow$ is image-finite). This allows to finally obtain the following characterization of observation equivalence, which is the most useful from a computational point of view, and will be used in the next Section:

Proposition 1.5.8

If a transition system is image-finite, then:

$$\approx \; = \bigcap_{k=0}^{\omega} \{F'^{k}(PxP)\} \quad \bullet$$

1.6. Observation equivalence for finite transition system

We show in this section that, for finite transition systems, the definition of observation equivalence given at the end of Section 1.5 directly provides an algorithm for equivalence verification.

Definition 1.6.1

A transition system $S(P, \Sigma, T, p_0)$ is **finite** if sets P (states) and Σ (observable actions) are finite.

•

A finite transition system is necessarily image-finite, hence for such system we have:

$$\approx \; = \bigcap_{k=0}^{\omega} \{\approx'_k\}.$$

Furthermore, since we have a non-increasing chain of relations, and all the elements of the chain are finite, there exists a k >=0 such that :

$$\approx'_k = \approx'_{k+1} = \approx .$$

An algorithm for checking the observation equivalence of two states p and q of a finite transition system is then:

Algorithm 1.6.2

```
R := PxP
while R > F' (R) do R := F' (R)
if (p, q)  in  R
        then write "p  ≈  q"
        else write  "not(p ≈ q)"
```

•

If we want to check the observation equivalence of two disjoint transition systems it is possible, and computationally convenient, to assume a starting relation, in Algorithm 1.6.2, smaller than $(P \cup Q)^2$. More precisely:

Proposition 1.6.3

Let $S_p(P, \Sigma_p, T_p, p_0)$ and $S_q(Q; \Sigma_q, T_q, q_0)$ be two transition systems. If Algorithm 1.6.2 is initialized with $R^0_{pq} := \{(p, q) \mid p \text{ in } P \text{ and } q \text{ in } Q\}$, and R^{fin}_{pq} is the relation computed by the algorithm, then $p_0 \approx q_0$ iff (p_0, q_0) in R^{fin}.

Proof. Any relation R over (P U Q) can be decomposed into four disjoint components:

$R_{pp}=\{(x, y) \text{ in } R \mid x \text{ in } P, y \text{ in } P\}$

$R_{pq}=\{(x, y) \text{ in } R \mid x \text{ in } P, y \text{ in } Q\}$

$R_{qp}=\{(x, y) \text{ in } R \mid x \text{ in } Q, y \text{ in } P\}$

$R_{qq}=\{(x, y) \text{ in } R \mid x \text{ in } Q, y \text{ in } Q\}$

It then suffices to realize that function *F'* operates in the well behaved way diagramatically illustrated in Figure 1.6.1. •

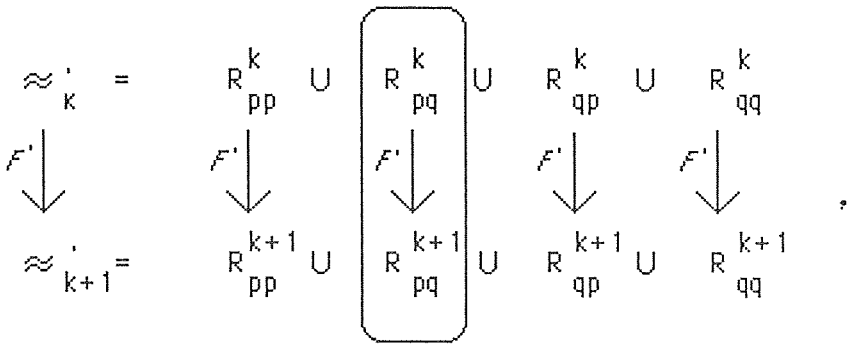

Figure 1.6.1 - The way function F' operates on relations.

The generic step included in the rectangle in figure is the one actually computed by the algorithm.

1.7. Testing-equivalence

We consider here another notion of equivalence, called testing-equivalence, denoted " $\cong$ ", and compare it with the k-equivalences introduced before. The interested reader will find more on testing-equivalence in [DeNH84].

Definition 1.7.1

If p in P is a state, $Q \subseteq P$ is a set of states and s in Σ^* is a string of observable actions, then

p **after** s = $\{p' \text{ in } P \mid p \Rightarrow p'\}$

Q **after** s = $\bigcup_{q \text{ in } Q} (q \text{ after } s)$

tau-closure(Q) = Q **after** ε(the empty word)

Definition 1.7.2

If $Q \subseteq P$ is a set of states and $L \subseteq \Sigma$ is a finite set of observable actions, then:

Q **MUST** L if $\forall q$ in tau-closure(Q). $\exists a$ in L. $=a\Rightarrow$ •

For the sake of clarity it is convenient to spell out Definition 1.7.2 for the limit case of empty Q

and/or L:

1) Q MUST Ø, with nonempty Q, means $\forall q$ in tau-closure(Q).($\exists a$ in Ø....)=$\forall q$....FALSE =FALSE

2) Ø MUST L, with nonempty L, means $\forall q$ in Ø. ... = TRUE

3) Ø MUST Ø means $\forall q$ in Ø. ... = TRUE

Definition 1.7.3

Let Q_1 and Q_2 be two sets of states. We say that Q_1 is **testing-equivalent** to Q_2, written $Q_1 \cong Q_2$, if:

$\forall s$ in Σ^*. $\forall$ finite L in Σ.

(Q_1 after s) MUST L iff (Q_2 after s) MUST L. •

Notice that for any set Q of states and any s in Σ^* we have: tau-closure(Q after s) = Q after s. Hence, for the purposes of the definition of ' $\cong$', taking the tau-closure of Q in Definition 1.7.2 (of MUST) is unnecessary.

Definition 1.7.4

Let p be a state. Then

traces(p) = {s in Σ^* | p=s$\Rightarrow$}

initials(p) = {a in Σ | p =a$\Rightarrow$ }. •

Notice that traces(p) = traces(q) iff p $\approx_1$ q.

For the purpose of comparing $\cong$ with $\approx_1$ and $\approx_2$ we reformulate the definition of the former. p $\cong$ q means:

$\forall s$ in Σ^*. $\forall$ finite L in Σ.

(p after s) MUST L iff (q after s) MUST L

that is:

$\forall p'$.(p=s$\Rightarrow$p' implies ($\exists a$ in L. p' =a$\Rightarrow$)) iff $\forall q'$.(q =s$\Rightarrow$ q' implies ($\exists a$ in L. q' =a$\Rightarrow$))

that is:

$\exists p'$.(p =s$\Rightarrow$ p' and initials(p')$\cap$L = Ø) iff $\exists q'$.(q =s$\Rightarrow$ q' and initials(q')$\cap$L = Ø).

Proposition 1.7.5

$p \approx_2 q$ implies $p \cong q$ implies $p \approx_1 q$ (or, $\approx_1 \geq \cong \geq \approx_2$).

Proof. Simply compare the three definitions below:

1) $p \approx_1 q$ means:

$\forall s$ in Σ^*.

$\exists p'.\ p=s\Rightarrow p'$ iff $\exists q'.\ q=s\Rightarrow q'$

2) $p \cong q$ means:

$\forall s$ in Σ^*. $\forall$ finite $L \leq \Sigma$.

$\exists p'.(p =s\Rightarrow p'$ and initials$(p') \cap L = \emptyset)$ iff $\exists q'.(q =s\Rightarrow q'$ and initials$(q') \cap L = \emptyset)$.

3) $p \approx_2 q$ means:

$\forall s$ in Σ^*. $\forall$ T in Σ^*.

$\exists p'.(p =s\Rightarrow p'$ and traces$(p')=$ T) iff $\exists q'.(q =s\Rightarrow q'$ and traces$(q')=$ T).

The ability to find equal sets of traces in (3) implies the ability to find equal sets of initials in (2). And (2) becomes (1) when L is fixed equal to $\emptyset$. •

Definition 1.7.3 of testing-equivalence implies the consideration of all strings of observable actions $\forall s$ in Σ^*...), with no bound on their lengths. Similarly to the case of observation equivalence, where alternative definitions based on bounded-length strings were chosen for deriving actual algorithms, we provide now an alternative definition of testing-equivalence based on action sequences of length at most 1.

Definition 1.7.6

Let P and Q be two sets of states.

a) $P \cong_0 Q$ is always true;

b) $P \cong_{n+1} Q$ iff

i)$\forall$ finite $L \leq \Sigma$. (P MUST L) iff (Q MUST L), and

ii) $\forall$ a in Σ. (P after a) $\cong_n$ (Q after a);

c) $P \cong Q$ iff $\forall$ n >= 0 $P \cong_n Q$.

The proof of the equivalence between this definition and Definition 1.7.3 is found in [DeN86].

2.1. Observation equivalence

We present here two algorithms for the verification of observation equivalence between finite labelled transition systems. The first algorithm ("refinements") is directly derived from the definition, and needs no proof of correctness. The second algorithm ("Sanderson") takes a different approach, and we provide the proof of its correctness here, since it is not found in the literature.

2.1.1. Algorithm "refinements"

We have developed an implementation of Algorithm 1.6.2., with the initialization suggested in Proposition 1.6.3, and called it "refinements". The environment chosen is LPA (Logic Programming Associates, Ltd) MacPROLOG, for the Macintosh personal computer. For an introduction to Prolog programming using LPA MacPROLOG see [CMcC84]. Figure 2.1.1.1 contains two snapshots of the Macintosh screen, showing the windows of program "refinements", the data window describing the most famous pair of non-observation-equivalent graphs, and the output window after a program run.

Window "**Comment**" is singled out: it gives a brief, semi-formal description of the algorithm (which is essentially Algorithm 1.6.2). The main advantage of using Prolog is that the actual program is almost a one-to-one translation of this description. In spite of the declarative nature of logic programming, the program has in part a procedural flavour: it iterates (via tail-recursion) the refinement of an equivalence (a list of pairs), until the pair of roots is excluded, or until refinement is no longer possible. A procedural style seems unavoidable in this case. Still, the use of a typically procedural language (e.g. Pascal) would have also required the implementation and handling of the data structures involved in the equivalence refinement, a burden completely eliminated by Prolog. Window "**Famous graphs**" in Figure 2.1.1.1 declares the nodes g_i (there are four) and h_j (five), represented as (g i)and (h j)), of the two graphs. It also provides the list of labelled edges of the two graphs (three and four, respectively), in the form: t(fromnode, label, tonode). The "**Default Output Window**" shows that after the second refinement the roots of the two graphs are found inequivalent.

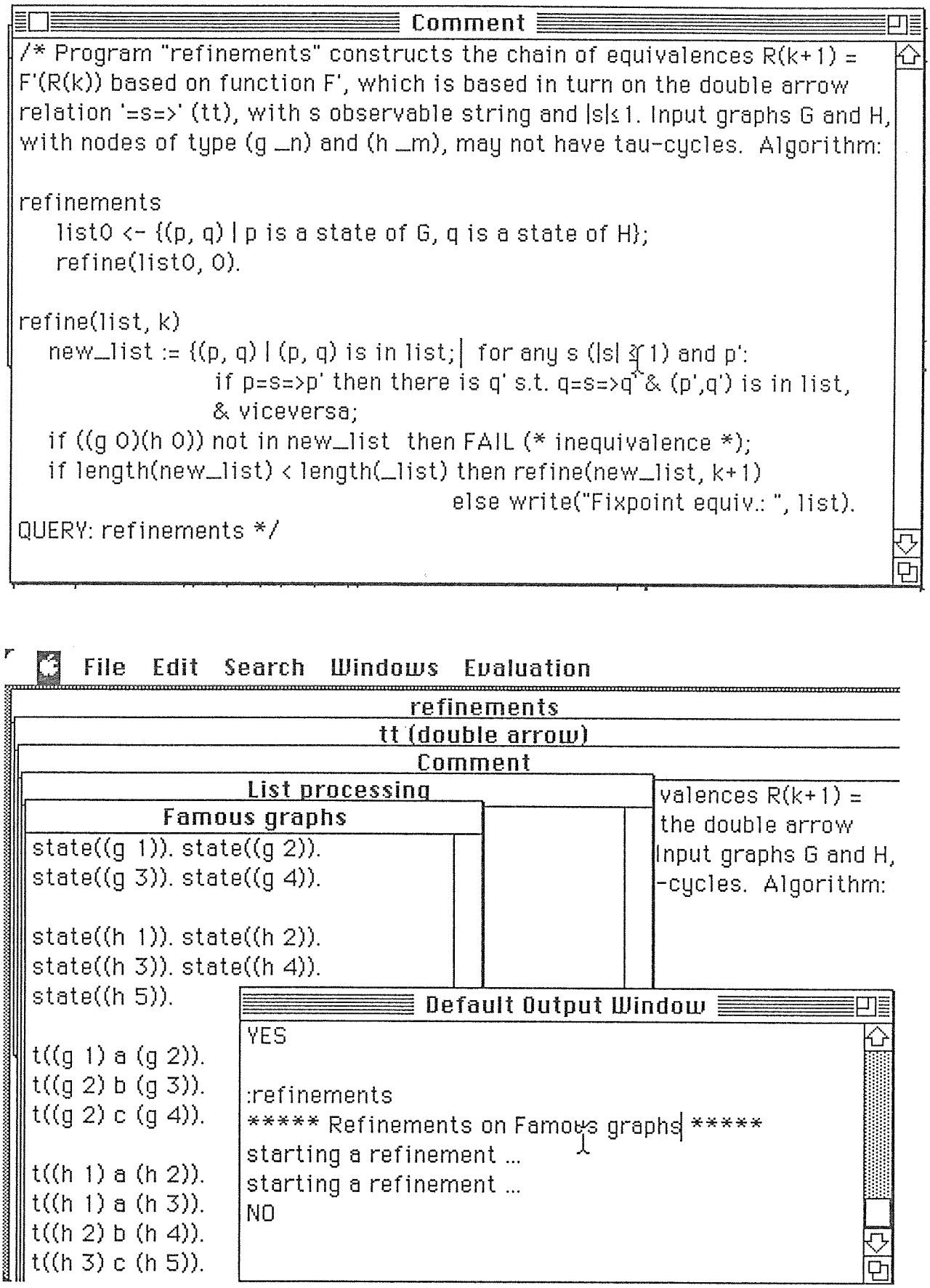

Figure 2.1.1.1 - Program "refinements"

As a slightly larger example, we have checked the observation equivalence of the two trees illustrated in Figure 2.1.1.2. They are derived from two alternative specifications, given in [BDN86], of a system that outputs a signal only after having received three input signals in any order.

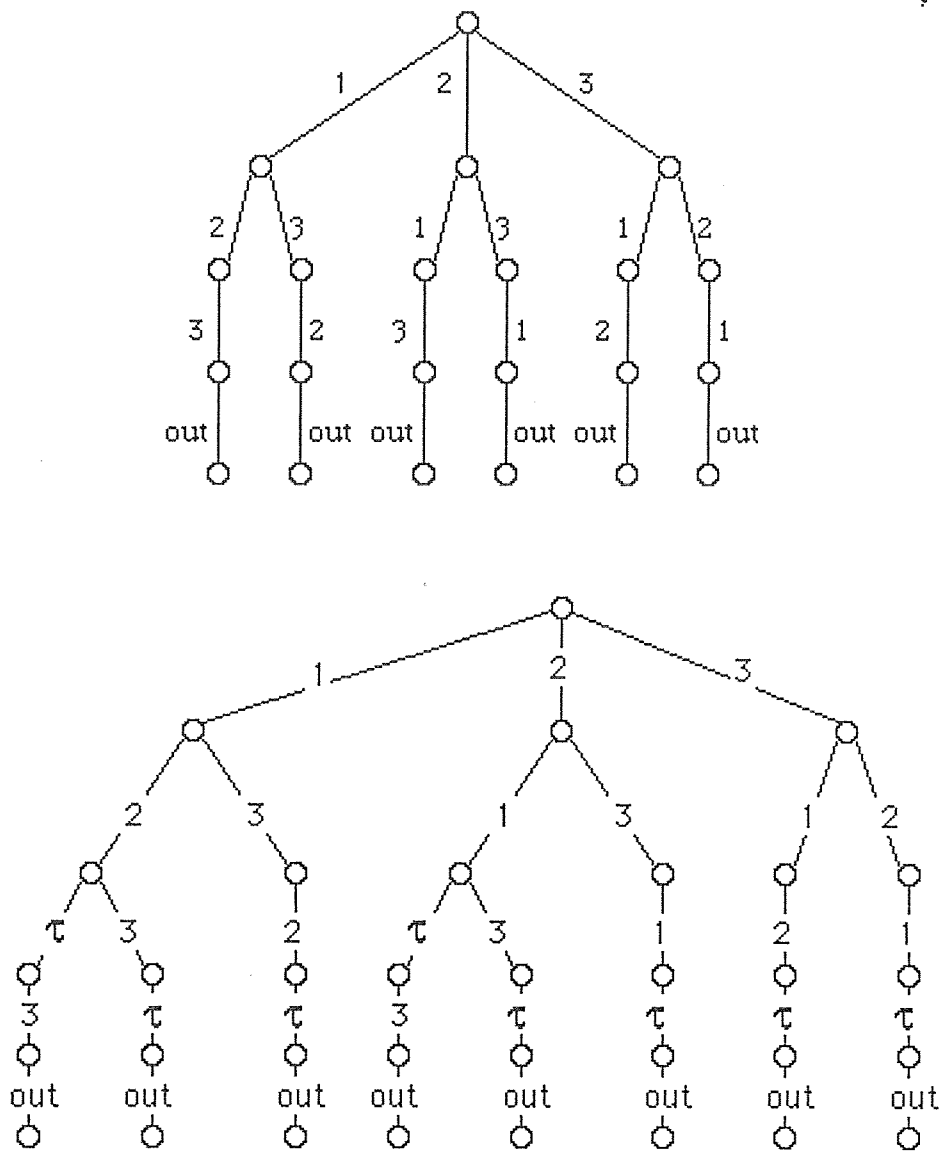

Figure 2.1.1.2 - Two observation equivalent trees

The fixpoint equivalence was found at the 2nd refinement. With 512 Kbytes of memory space, MacProlog took about 12 minutes (!) of Macintosh time to produce the output.

2.1.2 Algorithm "Sanderson" and its correctness

We have seen that for finite transition systems the definitions of observational equivalence via bisimulation (Section 1.3) and via the chain of k-equivalences (Section 1.4) coincide. Hence, as an alternative to the refinements strategy, one could prove the observational equivalence of two finite transition systems with roots p and q by building a bisimulation including pair (p, q). An algorithm for doing this is proposed in [San82], where a proof of correctness is unfortunately not provided. We will provide it here, but before even introducing the algorithm we need to extablish some facts.

Definition 2.1.2.1

State p of a transition system is **stable** if for any transition $p\text{-}x\rightarrow p'$ it is $x \neq \tau$. •

Definition 2.1.2.2

A transition system is **deterministic** if its transition relation $\text{-}\tau\rightarrow$ is empty (i.e. no transition is labelled by τ) and if whenever $p\text{-}x\rightarrow p'$ and $p\text{-}x\rightarrow p''$ then $p' = p''$.

Definition 2.1.2.3

Let P be the set of states and Σ be the alphabet of observable actions of a transition system. A relation $R \leq PxP$ is **well-constructed** if for any pair of states (p, q) in R the following three conditions are satisfied:

c1) $\forall q' \in P.\ (\ q\text{-}\tau\rightarrow q'$ implies $(p, q') \in R)$

c2) $\forall a \in \Sigma.\ \forall q' \in P.\ (\ q\text{-}a\rightarrow q'$ implies $\exists p' \in P.\ (\ p\text{-}a\rightarrow p'$ and $(p', q') \in R)$

c3) If q is stable then:

$\forall a \in \Sigma.\ \forall p' \in P.\ (\ p\text{-}a\rightarrow p'$ implies $\exists q' \in P.\ (\ q\text{-}a\rightarrow q'$ and $(p', q') \in R)$ •

Proposition 2.1.2.4

Let P and Q be the two state sets of two transition systems, also called, ambiguously, P and Q, and let R be a relation over PxQ. Assume that transition system P be *deterministic*, and that system Q be free of cycles of τ's. Then

R is well-constructed **implies** R is a bisimulation.

Proof

Once Proposition 1.5.2 is recalled, which states that the definition of bisimulation is independent of

the choice of function F or F', we must prove that

$\forall (p, q) \in R.$

$\forall s \in \Sigma^0 \cup \Sigma^1. \forall p' \in P.$

$p=s\Rightarrow p'$ implies $\exists q' \in P.$ ($q=s\Rightarrow q'$ and $(p', q') \in R$)

and viceversa.

Given the generic pair $(p, q) \in R$, we distinguish two cases.

i) $\mathbf{s = \varepsilon}$ (the empty string). Since P is deterministic, $p=\varepsilon\Rightarrow p'$ implies $p = p'$. Hence the corresponding transition sequence is $q=\varepsilon\Rightarrow q$, and $(p, q) \in R$.

Viceversa, if $q=\varepsilon\Rightarrow q'$, that is $q=q_0 \text{-}\tau\rightarrow q_1 \text{-}\tau\rightarrow \ldots \text{-}\tau\rightarrow q_n = q'$ $(n\geq 0)$, then the corresponding transition sequence is $p=\varepsilon\Rightarrow p$: since R satisfies condition **(c1)** in Def. 2.1.2.3, then $(p, q) \in R$ implies $(p, q_1) \in R$, and, inductively, $(p, q') \in R$.

ii) $\mathbf{s = a_1 \ldots a_n}$ $\mathbf{(n \geq 1)}$. Suppose $p=s\Rightarrow p'$. Since **P is deterministic** we may write

$$p = p_0 \text{-}a_1\rightarrow p_1 \quad \ldots \quad p_{n-1} \text{-}a_n\rightarrow p_n = p'.$$

The corresponding transition sequence $q=s\Rightarrow q'$, with $(p', q') \in R$, is found by generating a sequence of state pairs, starting at (p, q), whose elements inductively belong to R. The successor of the generic element (p_i, q_j) is obtained as follows: if q_j is not stable, then $q_j \text{-}\tau\rightarrow q_{j+1}$, and (p_i, q_{j+1}) is the new pair, which is in R by condition **(c2)** in Def. 2.1.2.3. If q_j is stable then, by condition **(c3)** transition $p_i \text{-}a_{i+1}\rightarrow p_{i+1}$ is paralleled by a transition $q_j \text{-}a_{i+1}\rightarrow q_{j+1}$, and the new pair (p_{i+1}, q_{j+1}) is also in R. Notice that the absence of τ–cycles in Q guarantees that a stable state in Q is always found in a finite number of transitions.

Viceversa, suppose $q=s\Rightarrow q'$. Conditions **(c1)** and **(c2)** are sufficient to guarantee that the existence of sequence $p\text{-}s\rightarrow p'$, with $(p', q') \in R$. •

Proposition 2.1.2.5

Let P and Q be two transition systems as in Proposition 2.1.2.4, and consider a relation R between their states. Then

R is a bisimulation **implies** R is well constructed.

Proof

We must show that any pair (p, q) in R satisfies conditions (c1) to (c3) of Def. 2.1.2.3. The proof is simple, and we only illustrate the case of **(c3)**: state q is stable, and we must show that whenever $p\text{-}a\rightarrow p'$, for some a and p', then $q\text{-}a\rightarrow q'$, for some q', and $(p', q') \in R$. If $p\text{-}a\rightarrow p'$, then $p=a\Rightarrow p'$, and since R is a bisimulation, $q= a \Rightarrow q''$, for some q". State q is stable, hence $q\text{-}a\rightarrow q' =\varepsilon\Rightarrow q''$, and also $q= a \Rightarrow q'$; again, since R is a bisimulation, $p = a \Rightarrow p''$, for some p", and $(p'', q') \in R$. But P is deterministic, thus p" = p'. In conclusion: $(p', q') \in R$.

We are now ready for a presentation of Sanderson's algorithm. Notice that indentation is used in the the description below bears semantic value.

Algorithm 2.1.2.6

```
SANDERSON
    checked := empty;
    to_be_checked := empty;
    APPEND((p0, q0), to_be_checked);
    while to_be_checked ≠ empty do
        x := FRONT (to_be_checked);
        if not GOOD(x) then FAIL;
        for-all y such-that GENERATES (x, y)
                if y ∉ checked and y ∉ to_be_checked then
                APPEND(y, to_be_checked);
    APPEND(x, checked);
    REMOVE-FRONT(to_be_checked)
SUCCESS
```

GOOD((p, q)) is TRUE iff:

$\forall a, q'.$ $(q\text{-}a\rightarrow q'$ implies $\exists$ p'. $p\text{-}a\rightarrow p')$

and

if q is stable then: $\forall a, p'.$ $(p\text{-}a\rightarrow p'$ implies $\exists$ q'. $q\text{-}a\rightarrow q')$.

GENERATES ((p, q), (p', q')) is TRUE iff:

$p\text{-}a\rightarrow p'$ and $q\text{-}a\rightarrow q'$, for some $a \neq \tau$

or

$p = p'$, and $\exists\, q'.\ q\text{-}\tau\rightarrow q'$.

Note

Variables "checked" and "to_be_checked" denote first-in-last-out queues. FRONT is a read-only operation, while operations REMOVE-FRONT and APPEND, respectively, remove the first element from the top and add a new element to the rear of the queue.

Proposition 2.1.2.7

Algorithm 2.1.2.6 always terminates.

Proof

If the algorithm fails, it terminates.

Otherwise we must prove that the number of iterations of the while loop is finite (with the number of steps per iteration being obviously finite too). Let n be the current number of pairs in "to_be_checked" plus "checked". At each iteration $k \geq 0$ new pairs are added to "to_be_checked", and one pair (x) is simply moved from it to "checked". Hence, n is increased or left unchanged. It is easy to realize that the invariant that all pairs in "to_be_checked" plus "checked" are distinct is preserved by the iterations. Since the number of possible distinct pairs is finite, the adding of pairs to "to_be_checked" terminates in a finite number of iterations; then the contents of this queue is moved, element by element, to the other queue, until the former is empty, and the loop successfully terminates.

Proposition 2.1.2.8

If algorithm 2.1.2.6 terminates with success, then $p_0 \approx q_0$.

Proof

We prove that after a successful termination of the algorithm, the relation in queue "checked", denoted R, contains (p_0 , q_0) and is well-constructed; thus, it is a bisimulation, by Prop. 2.1.2.4. Every pair which appears in queue "to_be_checked" during execution is eventually moved to queue "checked"; in particular, pair (p_0 , q_0) is initially in "to_be_checked", and is moved to "checked" at the end of the first iteration. We prove that R is well-constructed, i.e. that the generic element (p, q) of R satisfies conditions **(c1)** through **(c3)** of Def.2.1.2.3. Observe that since (p, q) appears in

"checked", it has played the role of x in some iteration of the while loop. We refer to this iteration.

Proof of **(c1)**. If q- $\tau\rightarrow$ q', then GENERATES((p, q), (p, q')), hence (p, q') is one of the y's appended to "to_be_checked" (if not already there) during the iteration, and eventually moved to "checked". Hence (p, q') $\in$ R.

Proof of **(c2)**. Since the algorithm did not fail, GOOD((p, q)). Hence, $\forall$a, q'. (q-a$\rightarrow$ q' implies $\exists$ p'. p-a$\rightarrow$ p'). Furthermore, GENERATES((p, q), (p', q')), thus (p', q') will also eventually appear in checked, and (p', q') $\in$ R.

Proof of **(c3)**. Similar to proof of **(c2)**.

Proposition 2.1.2.9

If algorithm 2.1.2.6 terminates with failure, then not($p_0 \approx q_0$).

Proof

For the purpose of contradiction assume $p_0 \approx q_0$. This fact would imply the existence of a bisimulation R which contains (p_0 , q_0). R would also be well-constructed (Prop. 2.1.2.5). If the algorithm fails, there must exist a chain (p_0 , q_0), (p_1 , q_1), ..., (p_n , q_n) such that GOOD((p_i, q_i)) and GENERATES((p_i , q_i), (p_{i+1} , q_{i+1})), for i = 0, 1, ..., n-1, and not GOOD((p_n , q_n)). On the other hand, if (p_0 , q_0)$\in$ R, and GENERATES((p_i , q_i), (p_{i+1} , q_{i+1})) for i = 0, 1, ..., n-1, then, inductively, (p_n , q_n)$\in$ R (the requirement that system P be deterministic is essential here). Finally, by definition of "well-contructed", any state pair that belongs to R, such as (p_n , q_n), must be GOOD, a contradiction. •

We have implemented algorithm 2.1.2.6, and called it "Sanderson". Again we provide in Figure 2.1.2.1. two snapshots of the Macintosh screen. Window "**Comment**" sketches the algorithm. "**Famous graphs**" is the same input window used with program "refinements". Notice that these two graphs satisfy the requirements for the applicability of the algorithm. The "**Default Output Window**" shows the tentative to construct a bisimulation relation, which fails after the generation of two pairs. No bisimulation exists, and the systems are inequivalent.

We have also applied this algorithm to the two trees of Figure 2.1.1.2. While the final refinement obtained by the previous algorithm on this example, in 12 minutes, contained 152 state pairs, the bisimulation found by this algorithm contains only 34 pairs, and is obtained in a few seconds.

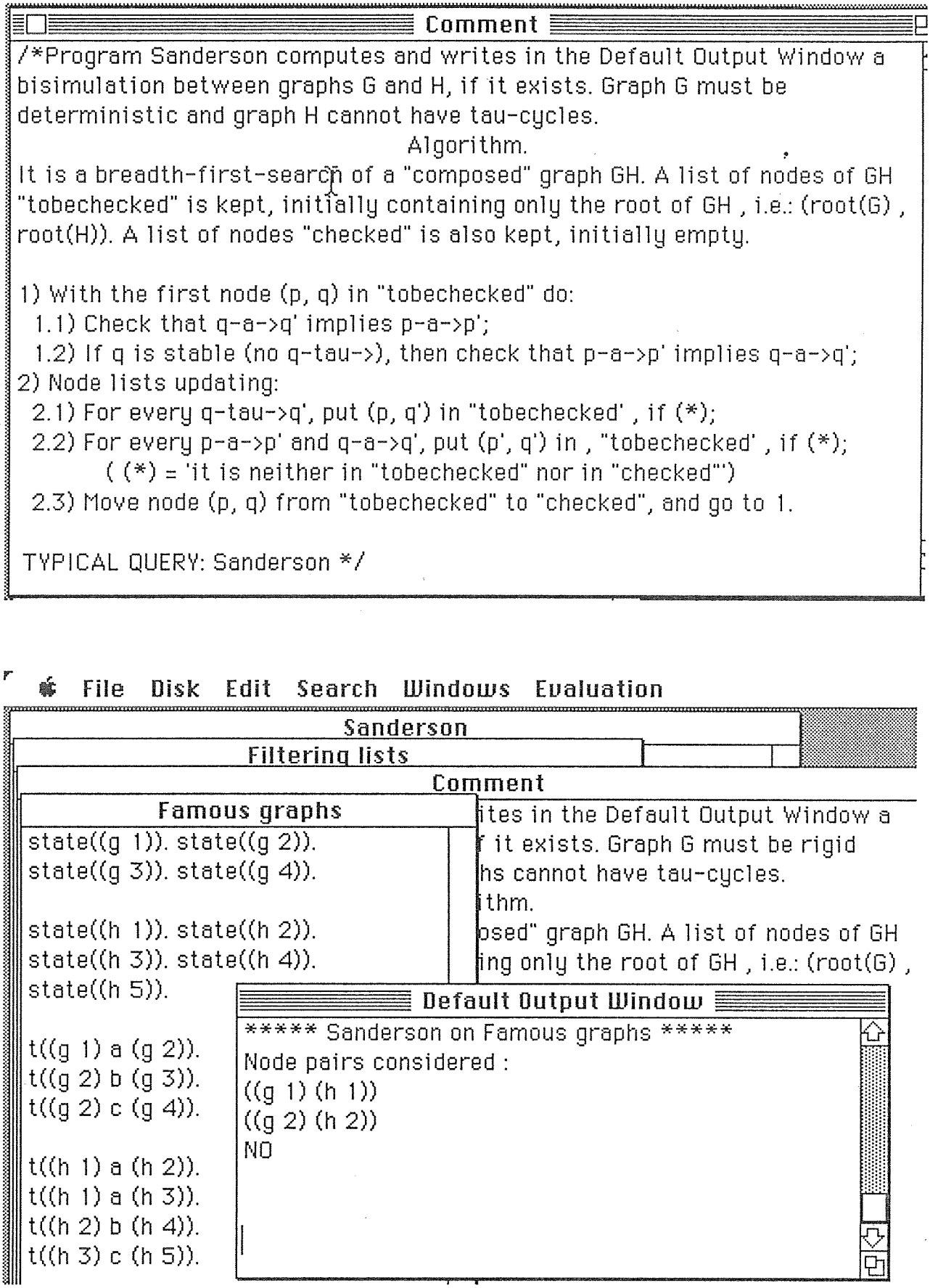

Figure 2.1.2.1 - Program "Sanderson"

2.2 Testing-equivalence

2.2.1 Algorithm "double subset construction"

We have based program "testing", for verifying the testing-equivalence of two finite transitio systems, on Definition 1.7.6. The algorithm, called "double subset construction", is outlined in Figur 2.2.1.1, window "**Comment**". The two basic functions of the algorithm are:

i) Transformation of the input graphs G and H, which are labelled and non-deterministic, int string equivalent deterministic graphs (say dG and dH). Each node of dX (X = G or H) is a subset of th nodes of X, and the root of dX is the tau-closure of the root of X. An arc P-a->P' is created for graph d if for some p in P and p' in P', p=a=>p' holds. This is the "subset construction" algorithm of [AU79].

ii) Computation of the "MUST-family" of a given node P of graph dX:

MUST-family(P) = {L $\leq \Sigma$ | P MUST L}.

The algorithm constructs graphs dG and dH "in parallel", which explains the name "double subse construction". (In fact this construction can be seen as a breadth-first-search generation/explorati of graph GHX = dG||dH, where '||' is the LOTOS parallel operator.) For each NODE (P, Q) of GHX, whe P is a node in dG and Q is a node in dH, the MUST-families of P and Q are compared for equalit (otherwise inequivalence is detected), for testing-equivalence, and the children of (P, Q) ar computed.

Again, program "testing" is tested on the pair of well known graphs in window "**Famous graphs**", an the negative result is reported in the "**Default Output Window**". Conversely, when the program i applied to the pair of trees of Figure 2.1.1.2, the answer is positive, as expected, and it is produced i about 100 seconds.

Comment

```
/*Program "testing" decides whether or not graphs G and H are
testing-equivalent. Both graphs cannot have tau-cycles.

                    Algorithm (Double subset construction)
It is a breadth-first-search of a complex graph GHX whose NODEs are pairs of
node subsets of G and H: the root is (tauclosure(root(G)), tauclosure(root(H))).
A list of NODEs of GHX "tobetested" is kept, initially containing only the root
of GHX. A list of NODEs "known" is also kept, initially empty.

1) With the first NODE (P, Q) in "tobetested" do:
    check that (P MUST L) iff (Q MUST L), for any L subset of observ. alphabet;
2) NODE lists updating:
 2.1) if (P after a)=P' and (Q after a)=Q', with 'a' observable action, and P'
    or Q' not empty, then put NODE (P', Q') in list "tobetested", if it  is neither
    in it nor in list "known";
 2.2) Move NODE (P, Q) from "tobetested" to "known", and go to 1.

QUERY: testing */
```

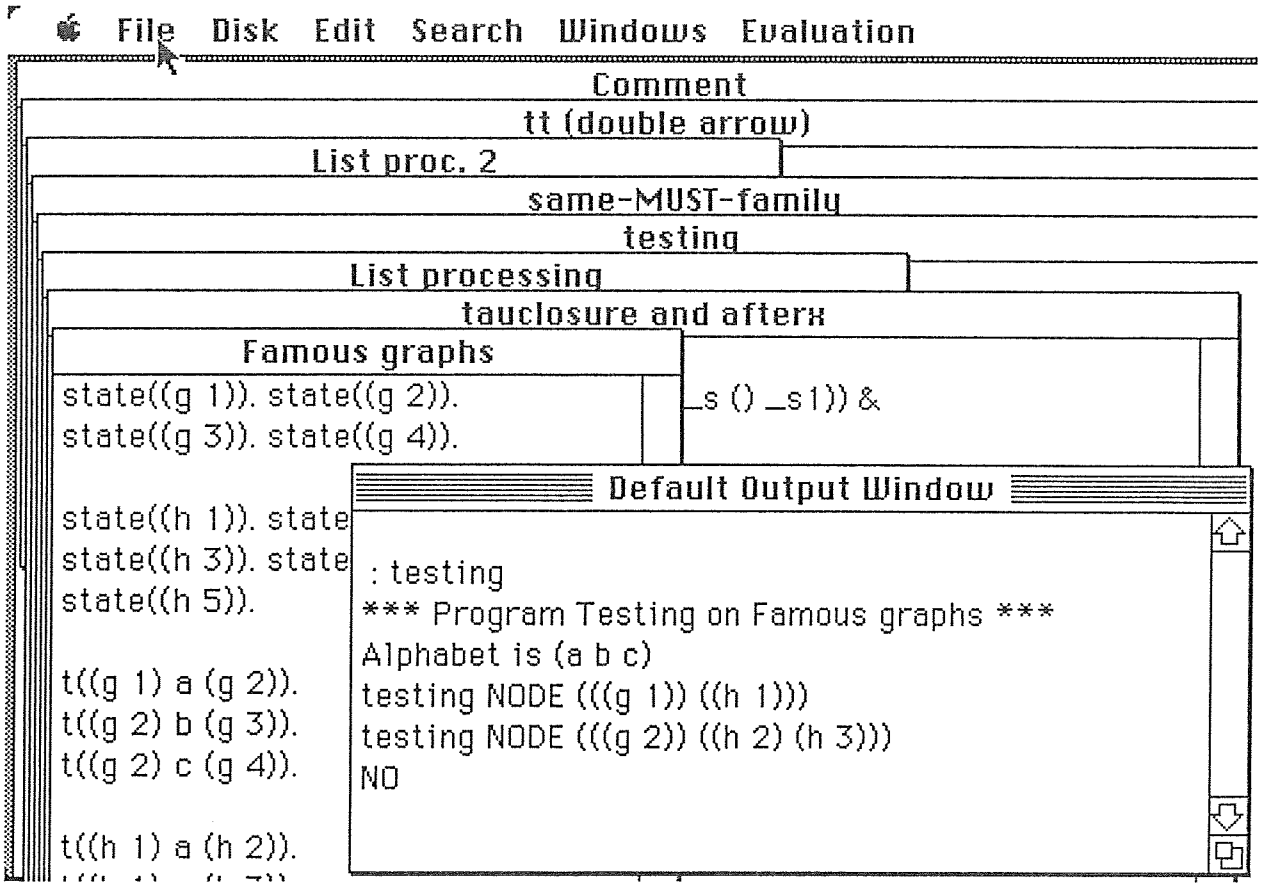

Figure 2.2.1.1 - Program "testing"

4. CONCLUSIONS

Some algorithms for equivalence-verification have been presented; their Prolog implementations have been briefly introduced, and applications to a small example have been illustrated. Prolog is convenient for a fast development of equivalence-verification algorithms. No difficulty should arise in developing Prolog verifiers for other relations between finite transition systems, such as the implementation relations discussed in [BSS86]. Computational efficiency is a different matter. Efficient algorithms are available for observational equivalence [KS83][PT86], and their implementation (not in PROLOG) seems a logical continuation of the work presented here. For testing-equivalence the picture is more complicated: the associated verification problem is of so called "intractable" complexity (PSPACE-complete).

An urgent development of this work is the integration of the algorithms discussed above into a more sophisticated tool which could compare for equivalence processes specified as **algebraic expressions** (e.g. in LOTOS), rather than graphs. Such a wider perspective obviously brings into the scene new difficulties, mainly due to the fact that expressions may or may not represent finite state processes, and that this problem is generally undecidable.

We wish to express our gratitude to Diego Latella, with whom many of the topics covered in Section 1 were discussed.

5. REFERENCES

[AU79] Aho, A.V., and J.D.Ullman, **Principles of Compiler Design**, Addison-Wesley, 1979.

[BDN86] Bolognesi, T., and R.De Nicola, A tutorial on LOTOS, CNUCE-C.N.R., Internal Report C86-7, Pisa, April 1986.

[BSS86] Brinksma, H., and G.Scollo, and C.Steenbergen, LOTOS Specifications, their Implementations and their Tests, Proceedings of the VI International Workshop on Protocol Specification, Testing, and Verification, Montreal, June 1986 (North Holland). Also: SEDOS/C2/N.40.

[CMcC84] Clark, K.L., and F.G.McCabe, **micro-PROLOG: Programming in Logic**, Prentice Hall, 1984.

[DeN86] De Nicola, R., Extensional equivalences for transition systems, internal report B4-41, I.E.I., C.N.R., Pisa, August 1986 (to appear in Acta Informatica).

[DeNH84] De Nicola, R., and M.Hennessy, Testing Equivalences for Processes, Theoretical Computer Science, Vol.34, 1984.

[DP8807] ISO, International Organization for Standardization/TC97/SC21 DP 8807, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, July 1986.

[HM85] Hennessy, M., and R.Milner, Algebraic Laws for Nondeterminism and Concurrency, Journal of the ACM, Vol.32, N.1, Jan. 1985.

[KS83] Kanellakis, P.C., and S.A.Smolka, CCS Expressions, Finite State Processes, and Three Problems of Equivalence, Proceedings of 2nd ACM Symposium on Principles of Distributed Computing, Aug. 1983.

[M80] Milner, R., A Calculus of Communicating Systems, Lecture Notes in Computer Science, Vol.92, Springer-Verlag, New York, 1980.

[M83] Milner, R., Calculi for Synchrony and Asynchrony, Theoretical Computer Science, Vol.25, 1983.

[PT86] Paige, R., and R.E.Tarjan, Three Partition Refinement Algorithms, draft.

[San82] Sanderson, M.T., Proof Techniques for CCS, Ph.D. thesis, University of Edinburgh, CST-19-82, 1982.