プログラミング 18－3
(1998. 3. 23)

テスト等価性に基づいた視覚的LTSモデル操作によるプロセス代数デバッガ

平手 孝　結縁 祥治　坂部 俊樹　稲垣 康善
名古屋大学大学院工学研究科情報工学専攻
hirate@sakabe.nuie.nagoya-u.ac.jp
{yuen,sakabe,inagaki}@nuie.nagoya-u.ac.jp

あらまし　本稿では、通信プロセスモデルにおいて、通信プロセスを視覚的にLTSで表現し、実現を表す通信プロセスの振舞いが仕様と等しくなるように修正するプログラミング環境としてデバッガ “VPAD” を提案する。ここで、Celikkan–Cleavelandのアルゴリズムを基礎に、相違情報をスタックへ保存しつつ、テスト等価性を判定する。等価でないとき、相違情報によって実現が仕様を満たさない状態を同定できる。我々は、通信プロセスの並行合成に対してこのアルゴリズムを拡張する。この拡張により、実現プロセスを構成する部分プロセス毎に修正すべき状態の指摘と行うことができる。視覚的に通信プロセスを表現することにより、ユーザに対して直接的に修正する状態を提示することが可能となった。

キーワード　プロセス代数、テスト等価性、受理グラフ、プログラミング環境

A Process Algebra Debugger for Testing Equivalences with visual LTS Manipulation

Takashi HIRATE　Shoji YUEN　Toshiki SAKABE　Yasuyoshi INAGAKI
Department of Information Engineering, Nagoya University
hirate@sakabe.nuie.nagoya-u.ac.jp
{yuen,sakabe,inagaki}@nuie.nagoya-u.ac.jp

Abstract　In this paper we present a visual debugger, “VPAD”, based on the communicating process model. We define “debugging” as changing an composite implementation to be equal to its specification by changing the sub-components of the implementation in the sense of the testing equivalence proposed by De Nicola–Hennessy's. The debugger automatically indicates which states of which components in an implementation do not satisfy its specification based on the diagnostic information generation of Cellikan–Cleaveland's. We have extended the algorithm for composite processes. VPAD is to make automated use of our debugging method where a user can interactively change a process behavior by a visual manipulation according to the distinguishing information.

key words　Process Algebra, Testing Equivalences, Acceptance Graph, Programming Environment

1 Introduction

The communicating process model has been actively studied and proposed as an abstract model for the concurrent computation. For theoretical purposes, a number of formal systems have been proposed such as CCS[Mil80, Mil89], CSP[Hoa85], ACP[BW90] and so on. A more practical approach can be seen as LOTOS[Bri86] and E-LOTOS[JTC98].

In the framework of the communicating process model, the verification on the abstract level is to establish an (algebraic) *behavioral relation* between a single process which represents an external behavior of the system and a set of subprocesses each of which represents a subcomponent of the system. The former process can be seen as a *specification* where the detailed system configuration is hidden. The latter can be seen as an *implementation* where the actual system configuration is represented in the description.

In this paper, we provide a development method of "debugging" the implementation process to be equal to the given specification. Since the implementation is given as a composition of subprocesses, modifying a subprocess may affect the interaction between subprocesses and unexpectedly change the behavior as a whole.

In [CC95], Celikkan and Cleaveland propose a diagnostic information generation if a pair of processes are not equal. In debugging processes, this information is quite helpful. But the algorithm is applied to a pair of simple processes, namely with no process composition. Since we need to handle the process composition for implementation processes, we propose an expansion of a composite process so that the expanded process can hold the location information of subprocesses. The location information is used to indicate which subprocess to be corrected.

Throughout the paper, we characterize process behaviors by the testing semantics[DH84] that is a "weak" semantics with a number of nice algebraic properties[Hen88]. By characterizing a process as an acceptance graph[CH93], the algorithm of the diagnostic information generation is applied for the testing semantics[CC95]. Although choosing this semantics is not essential to our method, we focus on developing the method from the practical point of view.

We present a visual process algebra debugger, called "VPAD", to make use of our debugging method. The authors have been developing a visual manupilation tool for processes in [HYSI97]. The visual manupilation is very useful in debugging compared to the symbol-based tools such as Concurrency Workbench[CPS93] in that (1) the state to be modified can be directly presented, and (2) the effect of the change can be easily understood by a user. The visual approach is also taken by [CLSS96] with more practical functions such as value-passing, while we intend to provide a generic graphical presentation for the existing verification tools.

The rest of the paper is structured as follows. Section 2 presents some preliminaries for the underlying theory. Section 3 extend the expansion to a composite process. In Section 4, we mention the diagnostic information generation for a composite process and present the debugging model. Section 5 illustrates our supporting tool "VPAD" Section 6 presents an example of application in VPAD. Section 7 concludes by some remarks.

2 Preliminaries

2.1 Processes, Tests and Preorders

Let a set of actions be $\mathcal{ACT}$ and assume the complementary operation over $\mathcal{ACT}$ as in CCS. Let the set of label $\mathcal{L} = \mathcal{ACT} \cup \overline{\mathcal{ACT}}$. τ denotes the completed action to represent a communication where $\tau \notin \mathcal{L}$.

Definition 1 (Process and Test) *A* process *is a labeled transition system* $P = \langle S_P, Act, \rightarrow_P, p_0 \rangle$ *where* S_P *is a set of states,* $Act \subseteq \mathcal{L}$, $\rightarrow_P \subseteq S_P \times Act \times S_P$ *is a labeled transition relation, and* $p_0 \in S_P$ *is an initial state. A* test *is a labeled transition system whose actions have the special action* w.

We write Act_w for $Act \cup \{\omega\}$ and Act_τ for $Act \cup \{\tau\}$. For $\langle s, \alpha, s' \rangle \in \rightarrow_P$, we write $s \xrightarrow{\alpha}_P s'$, $s \xrightarrow{\alpha}_P$ if $s \xrightarrow{\alpha} s'$ for some s', and $s \not\xrightarrow{\alpha}$ if $s \xrightarrow{\alpha} s'$ for no s'. In this paper, we deal only with finite LTS's where both $|S_P|$ and $| \rightarrow_P |$ are finite. A state s is *divergent* if there is an infinite τ-transition sequence from s, i.e. $s(\xrightarrow{\tau})^\omega$, written as $s \uparrow$. Otherwise, a state is convergent, written as $s \downarrow$[1]. We write $s \Uparrow$ if there exists some s' such that $s(\xrightarrow{\tau})^* s'$ and $s' \uparrow$. We write $s \Downarrow$, otherwise. $S \downarrow$ if for all $s \in S$ $s \downarrow$. A process (or a test) is deterministic if for all $s \in S$ and $a \in Act$, there exists at most one $s' \in S$ such that $s \xrightarrow{a} s' \in \rightarrow$.

[1] Note that these predicates are clearly decidable for a finite LTS.

Definition 2 (Experiment System) *Given an LTS $P = \langle S_P, Act, \rightarrow_P, p_0\rangle$ and a test $T = \langle S_T, Act_w, \rightarrow_T, t_0\rangle$, the experiment system of P by T, denoted by $\mathcal{E}(P,T)$, is a transition system $\langle S_P \times S_T, \rightarrow, \langle p_0, t_0\rangle\rangle$ where the transition relation is given as follows.*

- *For $a \neq \tau$, $p \xrightarrow{a} p'$ and $t \xrightarrow{a} t'$ imply $\langle p,t\rangle \rightarrow \langle p', t'\rangle$;*
- *$p \xrightarrow{\tau} p'$ implies $\langle p,t\rangle \rightarrow \langle p',t\rangle$; and*
- *$t \xrightarrow{\tau} t'$ implies $\langle p,t\rangle \rightarrow \langle p,t'\rangle$*

Given an experiment system $\mathcal{E}(P,T)$, a computation of $\mathcal{E}(P,T)$ is a maximal sequence of transitions from $\langle p_0, t_0\rangle$, i.e. the last state of which has no successive state, where p_0 and t_0 are the initial states of P and T respectively. A computation is *successful* if the sequence is finite and the test part of the last state has a transition labeled by w.

For $\mathcal{E}(P,T)$, we say P **may** T if $\mathcal{E}(P,T)$ has a successful computation and P **must** T if every computation of $\mathcal{E}(P,T)$ is successful.

Definition 3 (Testing Preorders)

*(1) $P \sqsubseteq_{\mathbf{may}} Q$ if for all $T \in \mathcal{T}$ P **may** T implies Q **may** T;*

*(2) $P \sqsubseteq_{\mathbf{must}} Q$ if for all $T \in \mathcal{T}$ P **must** T implies Q **must** T; and*

(3) $P \sqsubseteq_{\mathbf{test}} Q$ if $P \sqsubseteq_{\mathbf{may}} Q$ and $P \sqsubseteq_{\mathbf{must}} Q$

2.2 Acceptance Graph

As a fully abstract model of $\sqsubseteq_{\mathbf{test}}$, an acceptance tree is proposed[Hen85]. An acceptance graph is a graph representation of an acceptance tree. Following the literature[CH93], we call an acceptance graph an "Agraph". We say a set of action sets $A \subseteq 2^{Act}$ is minimal when for all $B \in A$ there is no $B' \subset B$ such that $A \in B'$ and $min(A)$ for the minimal set included in A.

Definition 4 (Agraph) *A labeled transition system $\langle T, Act, \rightarrow, t_0\rangle$ is an* Agraph *where each $t \in T$ is labeled by $t.acc \subseteq 2^{Act}$ and $t.conv \in \{true, false\}$ and the following conditions hold:*

1. *$\rightarrow$ is deterministic.*
2. *$s.acc$ is finite and each $A \in s.acc$ is finite.*
3. *$s.acc$ is minimal.*
4. *$s.conv = true$ if and only if $s.acc \neq \emptyset$.*
5. *$s.conv = false \wedge s \xrightarrow{a} s' \Rightarrow s'.conv = false$.*

We write $U^{\epsilon} = \{u' | u(\xrightarrow{\tau})^* u'$ for some $u \in U\}$ and $D(U, \alpha) = \{u' | u \xrightarrow{\alpha} u'$ for some $u \in U\}$ for a set of states U. For a state p, we write $S(p)$ for $\{\alpha | p \xrightarrow{\alpha}\}$. Given a process P, let $\mathcal{A}(P) = \langle T, Act, \rightarrow, t_0\rangle$ be an Agraph that satisfies the following conditions:

1. $T = \{\langle Q, b, A\rangle \mid Q \subseteq T, Q = Q^{\epsilon}, b \Rightarrow Q\downarrow\}$ where for $t = \langle Q, b, A\rangle$ $t.acc = A$ and $t.conv = b$.
2. For $t = \langle Q, b, A\rangle$, $A = \begin{cases} \emptyset & \text{if } s.conv = false \\ min(\{S(q)) | q \in Q, q \not\xrightarrow{\tau}\}) & \text{othewise} \end{cases}$
3. For $t_1 = \langle Q_1, b_1, A_1\rangle$ and $t_2 = \langle Q_2, b_2, A_2\rangle$, $t_1 \xrightarrow{\alpha} t_2$ only if:
   (a) $\alpha \neq \tau$; (b) $Q_2 = (D(Q_1, \alpha))^{\epsilon}$; and (c) $(b_2 \Rightarrow b_1) \wedge [(b_1 \wedge \neg b_2) \Rightarrow Q_2 \uparrow]$

Definition 5 (Prebisimulation) *Let Agraphs G_1 and G_2 and a relation Π between the states of G_1 and G_2. A relation R between the states of G_1 and G_2 is a prebisimulation when $R \subseteq \Pi$ and $\langle s_1, s_2\rangle \in R$ implies the followings;*

(1) $s_1 \xrightarrow{a} s_1' \Rightarrow \exists s_2' : s_2 \xrightarrow{a} s_2' \wedge \langle s_1', s_2'\rangle \in R$.

(2) $s_1.conv \Rightarrow s_2.conv$.

(3) $s_1.conv \wedge s_2 \xrightarrow{a} s_2' \Rightarrow \exists s_1' : s_1 \xrightarrow{a} s_1' \wedge \langle s_1', s_2'\rangle \in R$.

We write $s_1 \precsim s_2$ if there exists a prebisimulation containing $\langle s_1, s_2\rangle$. And we write $G_1 \precsim G_2$ if the initial states of G_1 and G_2 are prebisimular.

For a relation, R between the sates of G_1 and G_2, let $\mathcal{F}(R) = \{\langle p,q\rangle | ()p \xrightarrow{a} p' \Rightarrow \exists q'.q \xrightarrow{a} q' \wedge p'Rq']) \wedge (p \downarrow \Rightarrow [q \downarrow \wedge (q \xrightarrow{a} q' \Rightarrow \exists p'.p \xrightarrow{a} p' \wedge p'Rq')])\}$.

Theorem 1 *[CC95] Let a pair of Agraphs $G_1 = \langle S_1, Act, \rightarrow_1, s_{01}\rangle$ and $G_2 = \langle S_2, Act, \rightarrow_2, s_{02}\rangle$. Define $\precsim_0 = S_1 \times S_2$ and $\precsim_{k+1} = \mathcal{F}(\precsim_k)$. Then, $G_1 \precsim G_2$ iff $\forall k.s_{01} \precsim_k s_{02}$.*

This theorem indicates how to compute the prebisimulation preorder. Also it is easy to show that $\precsim_{i+1} \subseteq \precsim_i$ for all i.

In [CH93], it is shown that the testing preorder of processes is alternatively characterized by the prebisimulation preorder of the corresponding Agraphs starting with a certain relation.

Theorem 2 (Alternative Characterization) *[CH93] Let $\Pi = \{\langle t,u\rangle | t\downarrow$ implies $t.acc \subset\subset u.acc\}$ where $\mathsf{A} \subset\subset \mathsf{B}$ if for all $A \in \mathsf{A}$ there exists $B \in \mathsf{B}$ such that $B \subseteq A$.*

$$P \sqsubseteq_{\mathbf{test}} Q \text{ if and only if } \mathcal{A}(P) \precsim_\Pi \mathcal{A}(Q)$$

The Agraph is important for our tool design in that it provides a graphical representation fully abstract to the testing semantics.

3 Expansion of a Composite Process with Locations

A composite process is expanded to a single process to have a diagnostic information. Identifying a state to be corrected should enable to indicate which subprocess causes the problem. For this purpose, just expanding the composition operations as the expansion theorem of CCS[Mil89, Hen88] is not enough since it does not hold the subprocess location information.

Given a set of Agraphs $G_1, \cdots, G_n$ and a set of label L, we shall construct a single Agraph whose behavior represents the composition of $G_1, \cdots, G_n$ with the restriction of L.

The expansion is divided in two stages. We first construct an Agraph with communication information called a CAG (communicative Agraph) where the state transitions by the communications are distinguished in an Agraph. CAG is also a labeled transition system such that its behavior is obtained by its conversion to an Agraph. This intermediate structure is necessary in order to hold the communication causality that is not observable in a state of the composite Agraph.

Definition 6 (CAG) *Let each $G_i = \langle S_i, Act, \rightarrow_i, s_{0i}\rangle$ be an Agraph whose S_i is disjoint to each other and $L \subseteq Act$. $PLTS(G_1, \cdots, G_n, L)$ is a labeled transition system $\langle R, Act, \rightarrow, r_0\rangle$ with $R = \{\langle Q, A, b\rangle | Q = \{t_1, \ldots, t_n\}$ for $t_i \in S_i, A \subseteq 2^{Act}\}$ and $b \in \{true, false\}$. For each $r \in R$, $r.acc$ for A and $r.conv$ for b. We override r for Q if no confusion arises. In addition, the following condition must hold:*

1. $r_0 = \{s_{01}, \ldots, s_{0n}\}$
2. $r.conv = \bigwedge_{r' \in r.stat} r'.conv$
3. $r.acc = \begin{cases} \emptyset & \text{if } r.conv = false \\ min\{B \cup B' | B \in t.acc, B' \in t'.acc, t, t' \in r \text{ such that } t \neq t'\} & \text{otherwise} \end{cases}$
4. *For $r, r' \in R$, $r \xrightarrow{\alpha} r'$ exactly when:*
   (a) *For some $s \in r$, $s \xrightarrow{\alpha} s'$, $r' = (r - \{s\}) \cup \{s'\}$ and $\alpha \notin L$*
   (b) *For some $s_1, s_2 \in r.stat$ where $s_1 \neq s_2$, $\alpha = \tau$, $s_1 \xrightarrow{a} s_1'$, $s_2 \xrightarrow{\overline{a}} s_2'$, $r' = (r - \{s_1, s_2\}) \cup \{s_1', s_2'\}$*

A CAG G is again transformed into an Agraph $\mathcal{A}(G) = \{T, Act, \rightarrow_T, t_0\}$ where for $t \in T$ $t = \{r_1, \ldots, r_m\}$ and for each r_i, which is a state in G, $r_i = \{r_{i1}, \ldots, r_{in} | r_{ij} \in S_j\}$. We write $t.nodes$ for this $\{r_1, \ldots, r_m\}$.

4 Debugging a Composite Process

4.1 Diagnostic Information Generation

First we briefly mention the diagnostic information generation for the prebisimulation preorder[CC95]. Given Agraphs G_1 and G_2 where $G_1 \not\precsim G_2$, for some pair of states $\langle s_1, s_2\rangle$ of G_1 and G_2, the algorithms PREORDER and DFG shown in [CC95] produces a "witness" IHML[2] formula F such that $G_1 \models F$ but $G_2 \not\models F$. Then, the states to be corrected are those that do not satisfy F. To generate the formula, the prebisimulation preorder computation keeps track of non-prebisimular pairs of states at each stage accompanied with a reason causing non-bisimularity.

The reasons of non-bisimularity are listed by the definition of the preorder as follows:

[2] IHML stands for Intuitionistic Hennessy Milner Logic[Sti87].

0. $p \not\Pi q$ where Π is defined in theorem 2.
1. $p \xrightarrow{a} p'$ but $q \not\xrightarrow{a}$
2. $p \xrightarrow{a} p'$ and $q \xrightarrow{a}$ but $\forall q' : q \xrightarrow{a} q'$ and $p' \quad \not\precsim q'$
3. $p.conv = true$ but $q.conv = false$
4. $p.conv = true$, $q.conv = true$ and $q \xrightarrow{a} q'$ but $p \not\xrightarrow{a}$.
5. $p.conv = true$, $q.conv = true$ and $q \xrightarrow{a} q'$ but $\forall p' : p \not\xrightarrow{a} p' \Rightarrow p' \quad \not\precsim q]$.

For our purpose, it is not necessary to generate a witness formula since the implementation part of the non-prebisimular pair can be indicated on a graphical notation. We need a slight change to the algorithms in [?]. Figure 1 shows PREORDER. Figure 2 shows algorithm FINDDIFF whose input is the stack obtained by PREORDER and a pair of initial states. FINDDIFF returns the set of CAG states that cause non-bisimularity.

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

Figure 1: Computing the preorder

FINDDIFF$(Stack, p_0, q_0)$;
  $touple :=$ **TOP**$(Stack)$;
  $Stack :=$ **POP**$(Stack)$;
  if $\langle p_0, q_0\rangle$ **not in** *touple*
    then FINDDIFF$(Stack, p_0, q_0)$;
  else case *touple* of
    $[0, \langle p_0, q_0\rangle, err]$
      if $err \neq -1$ **then return**$(q_0.nodes)$;
      else
        $Q := \{q \mid (q \in q_0.nodes) \wedge (q.div = true)\}$;
        if $Q \neq \emptyset$ **then return**(Q);
        else
          return$(\{q' \mid (q \in q_0.nodes) \wedge (q \xrightarrow{\tau}{}^{+} q) \wedge (q \xrightarrow{\tau}{}^{*} q')\})$;
    $[1, \langle p_0, q_0\rangle, a, \langle p', -1\rangle]$
      return$(\{q \mid (q \in q_0.nodes) \wedge q \not\xrightarrow{a}\})$;
    $[2, \langle p_0, q_0\rangle, a, \langle p', q'\rangle]$
      FINDDIFF$(Stack, p', q')$;
    $[4, \langle p_0, q_0\rangle, a, \langle -1, q'\rangle]$
      return$(\{q \mid (q \in q_0.nodes) \wedge q \xrightarrow{a}\})$;
  end
end FINDDIFF;

Figure 2: Finding the incorrect states

Among the non-bisimularity reasons, reason 3 and 5 do not appear in the stack[CC93]. For reason 3, We start with Π as given in theorem2 and $\Pi = \precsim_0, \precsim_1, \ldots$ is a decreasing sequence. For reason 5, since an Agraph is deterministic, reason 2 and reason 4 imply reason 3. Thus, we omit those reasons from the original algorithm of PREORDER.

4.2 The Debugging Method

Figure 3 shows the model of debugging. Given a simple specification process and a composite implementation process as labeled transition systems, they are transformed into Agraphs. Then, for the implementation, CAG is generated. By transforming the CAG into an Agraph, we get the verification result. If the answer is YES (i.e. equal to the specification), then we finish the debugging. Otherwise, by FINDDIFF, the incorrect state in the implementation is identified. Referring to the corresponding CAG, we also can identify the incorrect states in the subprocess. A user correct one of the suggested corrections by the debugging tool, and repeats this procedure until the implementation reaches to a correct process.

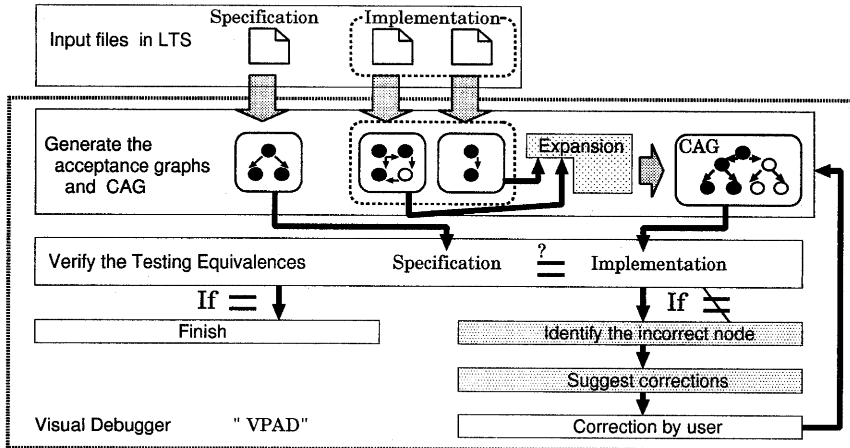

Figure 3: The Debugging Model

5 Graphical Debugger "VPAD"

We briefly present the configuration and the functions of our graphical debugger "VPAD"[3] written in Tcl/Tk. VPAD interact with a user by the following two kinds of windows.

One is called the *verification window* as shown in Figure 4, which presents the structures and attributes of Agraphs, debug command history, and verification results. A divergent state in an Agraph is colored as white, and a convergent state as gray. The Agraph of specification is shown on the left and that of implementation on the right. The Agrah shown in this window is the expanded Agraph where the incorrect state is highlighted by yellow. The closer information that correspond to this state is shown below these canvases. Lower on the right, verification results are shown.

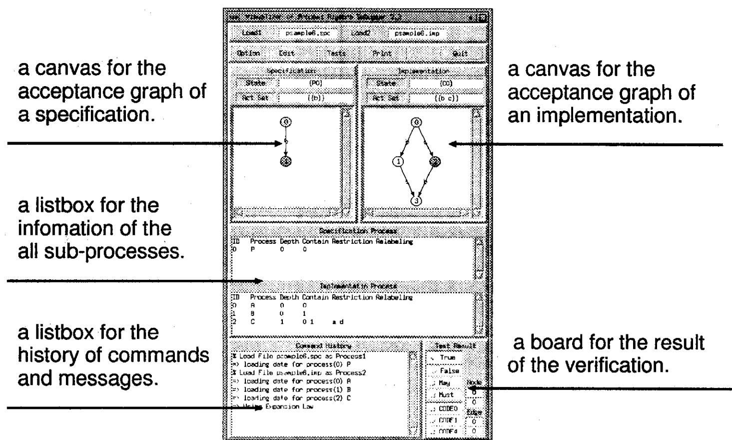

Figure 4: Verification window

The other is called the *organization window*, which displays the subprocess structure of the implementation. The incorrect states are highlighted also by yellow. These windows have a graphical edit function so that a user can change processes by visual manipulation. Figure 5 shows an example of the organization window where process 'C' shown in the right-most window is composed from processes 'A' and 'B' shown in the other two windows.

[3] VPAD stands for Visual Process Algebra Debugger

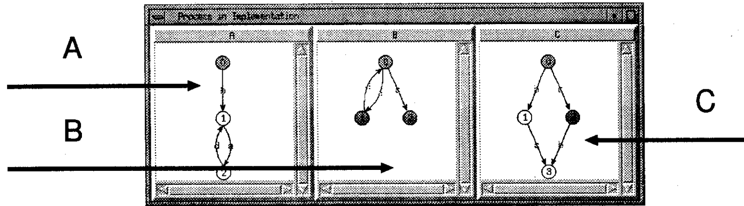

Figure 5: Organization window

6 An Example of A Debugging Step

We shall illustrate one step of the debugging method using VPAD as a small fragment of our method. While Let the specification and implementation be as Figure 6. Process 'C' is composed by two subprocesses 'A' and 'B'.

At this step, 'C' is diagnosed as having an incorrect extra transition labeled by 'c', which is not present in 'P'. Transition $0 \xrightarrow{c} 1$ in 'C' is known as originating in the transition $0 \xrightarrow{c} 2$ in the subprocess 'B'.

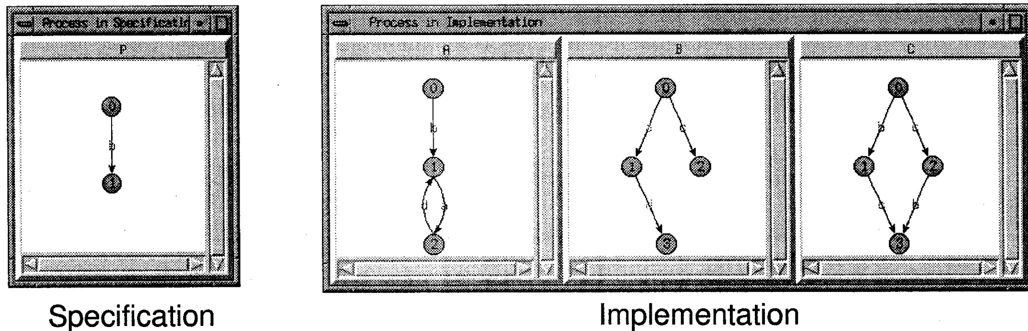

Figure 6: Agraphs before a correction

According to reason 4 in the stack, the debugger suggests deleting it by popping up a suggestion window as Figure 7.

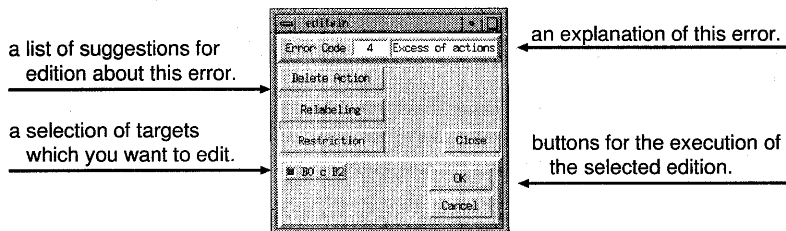

Figure 7: An example of suggestion window

The window has three suggestions. By selecting the 'Delete Action', a new frame appears at the bottom of this window. After this deletion, the graph of the implementation changes like Figure 8.

By deleting the transition, another transition labeled by 'c' is also deleted since there are two states of 'C' involved in this transition. But this completes matching the corrected 'C' to the specification.

7 Concluding Remarks

We proposed a graphical debugger "VPAD" which supports debugging a composite process to satisfy its specification given by a simple process. The debugger identifies the incorrect state based on the diagnostic information generation algorithms. In order to identify the incorrect states in the subcomponents, we

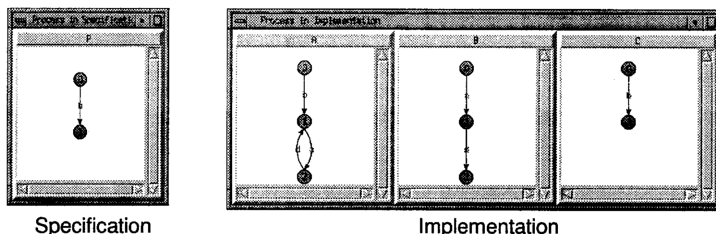

Figure 8: Agraphs after the correction

extended the Agraph structure so that the intermediate τ-transitions in a τ-closure of an Agraph are distinguished. VPAD suggests to a user what kind of corrections is appropriate using the diagnostic information. A user can interactively modify the subprocesses with the graphical interface of VPAD.

Applying our method to other semantics such as the bisimulation equivalences is a future work. Although this is not a big problem theoretically, the extension of VPAD is not obvious. In the current version of the debugger, we partially support the interface with the Concurrency Workbench NC[CS96]. We would like to extend the interface fully so that our tool can be used as an "intelligent" front-end for the symbol-based verification tools. We are also interested in using the similarity between specification and implementation as the guidance for efficient debugging.

References

[Bri86] E. Brinksma. A tutorial on LOTOS. In *Protocol Specification, Testing, and Verification V*, pages 171–194. North Holland, 1986.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1990.

[CC93] U. Celikkan and R. Cleaveland. Computing diagnostic tests for incorrect processes. 1993.

[CC95] U. Celikkan and R. Cleaveland. Generating diagnostic information for behavioral preorders. *Distributed Computing*, 9:61–75, 1995. The earlier version appeared in CAV '92.

[CH93] R. Cleaveland and M. Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5:1–20, 1993.

[CLSS96] R. Cleaveland, P. Lewis, S.A. Smolka, and O. Sokolsky. The concurrency factory: A development environment for concurrent systems. In *Computer Aided Verification '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 398–401, 1996.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transaction on Programming Languages and Systems*, 15(1):36–72, January 1993.

[CS96] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In *Computer Aided Verification '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, 1996.

[DH84] R. DeNicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Hen85] M.C.B. Hennessy. Acceptance trees. *Journal of the ACM*, 32(4):896–928, 1985.

[Hen88] M.C.B. Hennessy. *Algebraic Theory of Processes.* MIT Press Series in the Foundations of Computing. MIT Press, 1988.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice-Hall, 1985.

[HYSI97] T. Hirate, S. Yuen, T. Sakabe, and Y. Inagaki. A graphical debugger for labeled transition systems based on the diagnostic test generation. Technical Report COMP97-12, IEICE, May 1997. (In Japanese).

[JTC98] JTC1/SC21/WG7. *Enhancements to LOTOS.* ISO/IEC, 2 1998.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer, 1980.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Sti87] C. Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, 49:311–347, 1987.