Reducing Process Algebra

Victor Yodaiken
Copyright 2009.*
yodaiken@finitestateresearch.com

November 1, 2009

Abstract

Milner's "process algebra" is given a semantics in terms of "classical" automata.

1 Introduction

The following is representative of a common, but incorrect, argument made in the process algebra literature.

> *Basically, what is missing [in classical automata theory] is the notion of interaction: during the execution from initial state to final state, a system may interact with another system. This is needed in order to describe parallel or distributed systems, or so-called reactive systems. When dealing with interacting systems, we say we are doing concurrency theory, so concurrency theory is the theory of interacting, parallel and/or distributed systems.[Bae05]*

In this note, I will show how we can use *deterministic* automata to model process algebra processes. "Nondeterminism" can be modelled by relocating the locus of undetermined operation from the process to the environment

*Permission granted to make and distribute complete copies for non-commercial use but not for use in a publication. All other rights reserved but fair use encouraged as long as properly cited.

and specification. Concurrent composition of processes can be modelled in terms of automata products. The exercise illuminates some differences in perspective. In process algebra, one might say "when I run this concurrent program twice I get two different results because the program is intrinsically non-deterministic". In automata theory we can say "we get two different results because the deterministic program is not completely specified and depends on inputs from the environment." In process algebra, concurrency is the interleaving of processes via non-deterministic choice. In automata theory, the process algebra notion of concurrency looks more like a type of parallel composition that corresponds to an automata product. I'll use Milner's original formula [Mil79] as the reference process algebra just to use a well known base.

Section 2 introduces convenient method of working with state machines without getting bogged down in state. Section 3 covers the basic model and section 4 covers the "concurrent" construction. Although it can be readily seen that the basic operation of process algebra processes and composition are faithfully modelled, automata theoretic views are based on a very different notion of state and computation and the distinction may be of interest.

> What exactly is meant by the behavior of nondeterministic or concurrent programs is far from clear[MM85]

2 String functions

The traditional state-set/transition-map presentation of state machines can be supplemented or replaced by string functions

$$f : A^* \to X$$

where A^* is the set of finite strings over alphabet A including the empty string ϵ. The intuition is that $f(w)$ is the output of the transducer (Moore machine) represented by f in the state reached by following w from the initial state.

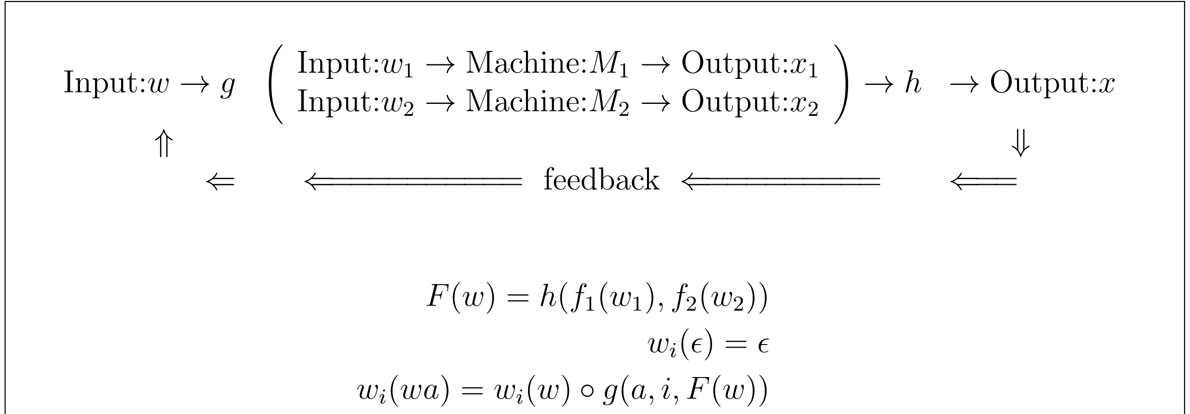
Input:w $\Rightarrow$ Machine:M $\Rightarrow$ Output:x

$$f(w) = x$$

The initial output of the transducer is given by $f(\epsilon)$. The transducers don't have to be finite, but of course string functions representing finite transducers are a key subclass. Details of this state machine presentation can be found in [Yod09] but, hopefully, the relationship between state machines and string functions is reasonably obvious. Clearly, a string function models a *deterministic* system by virtue of being a function and the challenges are to model the non-determinism that process algebra researchers emphasize as well as the composition and concurrent behavior of process. That is, we want first to describe what properties f must have to be a Milner process and then to show how the constructors like $a.P$ and $P+Q$ and $P|Q$ can be specified.

Process algebra processes can be considered to be string functions that output the set of events that are enabled in the current state. Milner suggests we think of these as buttons that can be pressed in that state. So $f(w) \subseteq A$ where A is the set of possible inputs. In fact, modulo a couple of other minor conditions, for $P : A^* \to X$ to be a "process" all we need is that $\forall w \in A^*, P(w) \subseteq A$. The process algebra constructors can then be treated as constraints and particular string functions can be seen to be solutions to those constraints.

String functions can often be specified in terms of recursive equations on the strings themselves. Let $w \circ z$ denote the concatenation of two strings. Contacenation of the empty string does nothing: $\epsilon \circ w \circ \epsilon = w$. If $\langle a \rangle$ is a string with a single element, I'll use a to abbreviate $\langle a \rangle$ and abbreviate $w \circ \langle a \rangle$ with wa and $\langle a \rangle \circ w$ with aw. Then $f(\epsilon) = x_0$ and $f(wa) = h(a, f(w))$ completely determines f. The process algebra "." operator can be described by something similar to $(a.f)(\epsilon) = a$, $(a.f)(aw) = f(w)$ — there are some details that have to wait until the next section. In a concurrent system we will have a system of connected string functions so that w corresponds to some w_i for each component f_i and wa corresponds to $w_i \circ z_i$ where z_i represents the effect of a on component f_i in the global state determined by w. This will all be made precise below.



$$F(w) = h(f_1(w_1), f_2(w_2))$$
$$w_i(\epsilon) = \epsilon$$
$$w_i(wa) = w_i(w) \circ g(a, i, F(w))$$

3 Processes and non-determinism

To explain non-determinism, Milner[Mil79] asks us to suppose we have two devices D_1 and D_2, so that when powered up D_1 has a possible a action followed by a choice of a b or c action. The D_2 device starts the same, but after the a action the state with b and c actions enabled can also non-deterministically jump to a state where only c is possible.

> *We do not know what determines the outcome for [D_2], perhaps it is the weather[Mil79]*

To accommodate this notion of non-determinism we relocate the non-determinism to the environment and to the specification. Relocation to the environment involves adding new inputs that can encode environmental signals which may change system state. Relocation to the specification involves permitting multiple "solutions" each with its own behavior.

Given Milner's example, suppose that the state machine "starts" by accepting an input from the weather or the current configuration of brownian motion in the universe, or the microseconds until Mercury is retrograde, or something else. That is, "powering up" has to be considered an event so that $D_2(\epsilon) = \{$ power up events$\}$ and we only get to a or b after power up. We turn on D_1 and D_2 by giving each an input from the set S of power up actions. Then $D_1(s) = \{a\} = D_2(s')$ for any $s, s' \in S$. But while $D_1(sa) = \{b, c\}$ all we know is that

$$D_2(s'a) \in \{\{c\}, \{b, c\}\}. \tag{1}$$

We relocate the non-determinism to the environment and distinguish D_1 which is oblivious to the environment for at least one step from D_2 which is not.

The power-up/weather/start input acts much like a "seed" in a pseudo-random number generator. If we can emulate random physical processes with such techniques, we can do the same for "non-deterministic" processes. The seed could be an unbounded integer or a complex number or a 2048 bit binary string, but we want to be able to know that if s is a seed then $f(sw)$ depends on s in some unspecified way. In fact, the "amount" of non-determinism is then a function of the size of S. If you want a lot of non-determinism make S be the set of real numbers, for example.

As a second use of relocation to the environment, we can suppose that there is a time event ρ not otherwise in A that we make enabled in every state. Input of this event represents the passage of a unit of time. Let ρ^n be a string of n ρ's. The specification above has D_2 transitioning to either a $\{c\}$ or a $\{b,c\}$ state depending on the seed and some unknown internal computation. Let's now also let D_2 jump from a state where $\{b,c\}$ is enabled to a set where only c is enabled without anything happening except time passing and make D_1 impervious to time $D_1(sa\rho \circ u) = D_1(sa \circ u)$ but allow D_2 to change state when time passes. The constraint:

$$D_2(s'a\rho^{n+1}) \subseteq D_2(s'a\rho^n) \tag{2}$$

allows D_2 to disable b just because time has passed. In fact, there is no reason to limit ourselves to S and $\{\rho\}$, we could add additional inputs to represent changes in temperature or magnetic flux or the expansion of the universe or just some unspecified "environmental" effect. The idea however is to consider the device deterministic, but subject to unspecified effects from external signals. As seen in the next section, this model extends to concurrent composition and it also fits in well with treating process string functions as solutions to constraints — something that allows for even more undetermined behavior. It may be that there is some fundamental reason to want to consider "non-determinism" as an intrinsic system property rather than as the effect of the environment, but that reason is not articulated in any of the process algebra literature I have seen.

Definition 3.1 *P is a time sensitive process on alphabet A and random seeding S if and only if:*

$$\{\rho,\tau\} \cap (S \cup A) = A \cap S = \emptyset \tag{3}$$

$$
\begin{aligned}
\textit{and } P(\epsilon) &= S && (4)\\
\textit{and } P(sw) &\subseteq (A \cup \{\rho\}) && (5)\\
\textit{and } P(sw) \neq \emptyset &\rightarrow \rho \in P(sw) && (6)
\end{aligned}
$$

What happens if $a \notin P(w)$ and we go to $P(wa)$? We just leave that unspecified.

Now we can make the solution to D_2 precise.

$$
\begin{aligned}
&\text{A process } P \text{ with alphabet } A \text{ and seed set } S\\
&\qquad\text{is a solution to } D_2 \text{ if and only if}\\
&\qquad P(s) = \{a\} && (7)\\
&\qquad P(sa\rho^n) \in \{\{b, c\}, \{b\}\} && (8)\\
&\qquad P(sa\rho^{n+1}) \subseteq P(sa\rho^n) && (9)
\end{aligned}
$$

$$
\begin{aligned}
&\text{A process } P \text{ with alphabet } A \text{ and seed set } S\\
&\qquad\text{is a solution to } D_1 \text{ if and only if}\\
&\qquad P(s) = \{a\} && (10)\\
&\qquad P(sa\rho^n) = \{b, c\} && (11)
\end{aligned}
$$

The constructors $a.P$, $P + Q$, and $P\ L$ - are now simple. Suppose P and Q are processes on S and A_P and A_Q then:

- If $a \in A_P$ and $Q = (a.P)$, then $Q(s) = \{a\}$ and $Q(saw) = P(sw)$.
- R is a solution to $P + Q$ if and only if there is some function *choice* so that $\mathit{choice}(s, a) \in 1, 2$ and if $\mathit{choice}(s, a) = 1$ then $a \in A_P$ and if $\mathit{choice}(s, a) = 2$ then $a \in A_Q$ and $R(saw) = P(saw)$ if $\mathit{choice}(s, a) = 1$ and $R(saw) = Q(saw)$ otherwise. We don't need to specify anything about *choice* other than that it must pick P or Q and if the event a only belongs to one of A_P or A_Q that it must pick the respective process.
- if $L \subset A_P$ and $Q = P \setminus L$ then using $\setminus$ as the standard set subtraction:

$$
Q(swa) = \begin{cases} P(swa) \setminus L & \text{if } a \in Q(sw);\\ \emptyset & \text{otherwise.} \end{cases}
$$

4 Composition and concurrency

For $P|Q$ we need to think about intended semantics which are apparently inspired from a view of threads running under a time-sharing scheduler. The idea is that at each step one of three things happens: P advances by accepting an input from the environment, Q advances by accepting an input from the environment, or both components advance by matching inputs while the pair accepts an anonymous input τ from the environment. For the last, we suppose that there are inputs $a \in A_P$ and $\bar{a} \in A_Q$ which are known to match. Let $\bar{\bar{a}} = a$.

The automata based model of concurrency comes out of a completely different idea of how to think about parallel processing and is based on automata products and feedback. In the general automata product the state set of the product is the vector of states of the factor machines. If we have n factor state machines, the product vectors $\vec{d} = (d_1, \ldots d_n)$ produce output vectors $(x_1 \ldots x_n)$ by application of the output maps of each factor machine to the appropriate element. Interaction between machines comes about via the feedback. Write write $a(d)$ for the state reached by following a from state d and extend that to strings so $\epsilon(d) = d$ and $wa(d) = w(a(d))$. For a product, an input a is mapped to a sequence of inputs for each factor using the outputs of the factors as parameters. If $\vec{d}$ is the current state and $\vec{x}$ is the vector of outputs then $(a, \vec{x}, i) \mapsto z_i$ a sequence of events and $a(d) = (z_1(d_1) \ldots z_n(d_n))$. We can structure this mapping so that, for example, if x_i indicates component i is sending a message m to component j, the string z_j causes j to receive the message. Such products can get rather involved, but they can be described concisely terms of simultaneous recursion on string functions without worrying about state vectors.

In the context of string functions a general product is described as $n+1$ functions f (the composite), and a sequence valued function w_i for each factor f_i that depends on w and the feedback. Details of the product can be found in [Yod09] but I'll just use a form suited to process composition here.

Suppose P_1 and P_2 are defined string function processes. Then Q is a solution to $P_1|P_2$ only if Q can be constructed as product of P_1 and P_2 with the appropriate interleaving of inputs and proper handling of τ. There is a great deal of implicit complexity in the process algebra notion of concurrent composition — for example, if Q is a solution to $P_1|(P_2|P_3)$ there should be possible τ transitions that do not cause a state change in P_1. All that complexity has to be accounted for when we make things concrete in string

functions. For the definition below, let $i' = 1$ if $i = 2$ and let $i' = 2$ if $i = 1$.

Definition 4.1 *Q is a solution to $P_1|P_2$ where P_1, P_2 are processes if and only if:*

$$\textit{There is some } F(w) = (P_1(w_1), P_2(w_2)) \tag{12}$$

$$\textit{so that if} F(w) = (X_1, X_2)$$

$$\textit{then } Q(w) = \begin{cases} X_1 \bigcup X_2 \bigcup \{\tau\} & \textit{if } \exists b \in X_1 : \bar{b} \in X_2 \\ X_1 \bigcup X_2 & \textit{otherwise.} \end{cases} \tag{13}$$

$$\textit{and } w_i(\epsilon) = \epsilon \tag{14}$$

$$\textit{and } w_i(< s >) =< s > \tag{15}$$

$$\textit{and } w_i(swa) = w_i(sw) \circ z_i \tag{16}$$

$$\textit{where if } a \in Q(sw) \tag{17}$$

$$z_1 = \langle \rho \rangle = z_2 = \langle a \rangle \tag{18}$$

$$\textit{or } a = \tau \textit{ and } z_i = \langle b \rangle \textit{ and } \langle \bar{b} \rangle = z_{i'} \textit{ for } b \in X_i, \bar{b} \in X_{i'} \tag{19}$$

$$\textit{or } a = \tau \textit{ and } z_i = \langle \tau \rangle \textit{ and } z_{i'} = \epsilon \textit{ and } \tau \in X_i \tag{20}$$

$$\textit{or } a \in A_{P_i} \textit{ and } z_i = \langle a \rangle \textit{ and } z_{i'} = \epsilon \textit{ and } a \in X_i \tag{21}$$

$$\tag{22}$$

Condition 13 defines $Q(w)$ to output the union of the two elements of $F(w)$ (the output of the P_1 factor the output of the P_2 factor) with τ added if there is a matching enabled input. Condition 16 defines the connection to generate on each step a z_i for each factor P_i with the properties given in condition $17 - 21$. Those properties are that if $a \in Q(w)$ (a is enabled) the ρ inputs are just passed in parallel to the factors (condition 18), τ can cause the two factors to advance in parallel with a matching transition (condition 19), or τ can cause one factor to advance when it enables τ directly (condition 20), and finally that ordinary inputs are passed to one of the factors. Note that F is not a process. We construct F from P_1 and P_2 and then construct Q from F.

Clearly, there can be any number of solutions to such composition. I think it's reasonably obvious that this definition satisfies the conditions in Milner's construction but it exposes some oddities. The requirement to pass τ down the hierarchy was not obvious to me from Milner's rules, but if P_1 is a solution to $P_3|P_4$ and $P_1(sw_1) = \{\tau\}$ then $Q(sw)$ must include τ no matter whether P_2 and P_1 are ready to communicate if the basic rules of process algebra are

going to hold. More interesting is the question of conditions for liveness. If z_i is solely a function of a and $F(w)$ then "liveness" is impossible to guarantee unless P_1 and P_2 always get to states where a matching communication is the only possible event. For example, if P_1 will output b waiting for a $\bar{b}$ to become enabled on P_2 but eventually times out after some number of ρ's then there is no way for the computation of z_i to make sure both sides advance because there is no state information about the number of times that P_i has been passed up.

There seems to be no way to solve this problem within process algebra itself although *fairness* is a standard requirement for schedulers. Actually, it's not hard to fix the semantics by making the product slightly more complex. We can add an additional component to the product: to track changes in which of P_1 or P_2 was picked last and for how many consecutive operations that component has been selected.

$$How(\epsilon) = (0,0) \tag{23}$$

$$How(wa) = \begin{cases} (i, n+1) & \text{if } How(w) = (i,n) \text{ and } a = i \\ (i,1) & \text{if } a = i > 0 \text{ and } How(w) = (j,m) \text{ for } j \neq i \\ How(w) & \text{otherwise.} \end{cases} \tag{24}$$

$$\text{Let } How(w)(thread) = i \text{ if } How(w) = (i,n) \tag{25}$$

$$\text{Let } How(w)(time) = n \text{ if } How(w) = (i,n) \tag{26}$$

Definition 4.2 *Q is a t* fair *solution to $P_1|P_2$ where P_1, P_2 are processes if and only if:*

$$\textit{There is some } F(w) = (P_1(w_1), P_2(w_2), How(w_3) \tag{27}$$

$$\textit{so that } if F(w) = (X_1, X_2, x_3)$$

$$\textit{then } Q(w) = \begin{cases} X_1 \bigcup X_2 \bigcup \{\tau\} & \textit{if } \exists b \in X_1 : \bar{b} \in X_2 \\ X_1 \bigcup X_2 & \textit{otherwise.} \end{cases} \tag{28}$$

$$\textit{and } w_i(\epsilon) = \epsilon \tag{29}$$

$$\textit{and } w_i(< s >) =< s > \tag{30}$$

$$\textit{and } w_i(swa) = w_i(sw) \circ z_i \tag{31}$$

$$\textit{where if } a \in Q(sw) \textit{ and } i \in 1,2 \tag{32}$$

$$z_1 = \langle \rho \rangle = z_2 = \langle a \rangle \textit{ and } z_3 = \epsilon \tag{33}$$

$$\textit{or } a = \tau \textit{ and } z_i = \langle b \rangle \textit{ and } \langle \bar{b} \rangle = z_{i'} \textit{ for } b \in X_i, \bar{b} \in X_{i'}$$

$$\textit{and } z_3 = \langle i \rangle \tag{34}$$

$$\begin{aligned}
or\ a = \tau\ and\ z_i = \langle\tau\rangle\ and\ z_{i'} = \epsilon\ and\ \tau \in X_i\ and\ z_3 = \epsilon \qquad (35)\\
or\ a \in A_{P_i}\ and\ z_i = \langle a\rangle\ and\ z_{i'} = \epsilon\ and\ a \in X_i\\
and\ z_3 = \langle i\rangle\ and\ (a \notin X_{i'}\ or\ x_3(thread) = i'\ or\ x_3(time) < t) \qquad (36)
\end{aligned}$$

References

[Bae05] J. C. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335, 2-3:131–146, May 2005.

[Mil79] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

[MM85] Hennessy M. and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32, 1:137–161, January 1985.

[Yod09] Victor Yodaiken. Primitive recursive transducers. Technical report, Finite State Research LLC, August 2009. `http://www.yodaiken.com/papers/recursivetransducer.pdf"`.