Reducing Process Algebra

Victor Yodaiken

yodaiken@finitestateresearch.com

May 7, 2010

Abstract

Milner's "process algebra" is given a semantics in terms of deterministic state machines.

1 Introduction

The following incorrect claim is not unusual 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]*

Actually a sophisticated notion of state machine product was developed for representing composition of "interacting" state machines starting in the 1950s[HS66]. A general survey can be found in a monograph by Gecseg[Gec86], Domoni provides a more modern, more algebraic treatment [DN04] and [Yod09] provides practical techniques for construction of complex products.

In this note, I show how deterministic state machines machines modeling "process-algebra like" systems such as Milner's original formulation[Mil79] can be defined and how to define constructions that emulate the process order constructors $a.P$ and $P + Q$ and $P|Q$ and $P \setminus X$. For every "rule" of the form $P \xrightarrow{a} P' \vdash Q \xrightarrow{a} Q'$ we will have a provable statement for the form "P enables a and after a P behaves like P'". The very triviality of these results illustrates something about the process algebra.

The reader familiar with the process algebra literature will note that such an effort must find a way to model the non-determinism that is so fundamental to the process algebra world-view. Since deterministic programs are used to produce pseudo-random number sequences and to model Brownian motion and even the stock market (perhaps not the best example at this date) the problem is easier to solve than it appears at first. Section 4 will include a short discussion on the difference between "real" non-determinism and simulated non-determinism, but certainly there is nothing in the basic axiom set of Milner's original process algebra that notices this distinction as far as I can see.

Section 2 sketches the emulation in terms of traditional state-set presentations of state machines and introduces fake non-determinism. Section 2.2 adds concurrent composition. Section 3 then makes everything precise in terms of string functions, which simplify the treatment of composition. And section 4 includes a short discussion on the meaning of it all.

2 State machines and process algebra processes

The purpose of this section is to build some intuition in the familiar grounds of state machines as sets of states with transition and output maps.

2.1 Outputs and faking non-determinism

The classical definition of a Moore type state machine is a 6-tuple:

$$M = (A, X, S, start, \delta, \gamma)$$

where A is a set of inputs, X is a set of outputs, S is the set of states, $start \in S$ is the start or initial state, $\delta : S \times A \to S$ is the transition function,

and $\gamma : S \to X$ is the output function. It is usual to require A and S to be finite, but I am not doing that since process algebra does not embrace that limitation of actual computing devices and programs.

Let A be the universal set of transitions for a collection of process algebra processes. The output of a state machine emulating a process algebra process will be the set of enabled inputs. An output of $\emptyset$ means the process is stuck. Let R be an alphabet of "random seeds" that we can adjoin to the original universal input alphabet A and say M is R pseudo-random if the initial input for M can be any element of R, that is if $\delta_M(start, r)$ is defined for every $r \in R$ where *start* is the initial state of M. The extent of non-determinism can be varied by varying the cardinality and complexity of R. For example, if R is the set of real-numbers, then R injects a robust $\aleph_1$ pseudo-randomness into M. Consider the events in R to be "start up" events.

Definition 2.1 $M = (A, X, S, start, \delta, \gamma)$ *is a* process algebra compatible state machine *with input alphabet A and random seed alphabet R iff $\delta(s, a)$ is complete (defined on all pairs of arguments) and*

$$\gamma(start) = R \tag{1}$$

$$If\ s \neq start\ then\ \gamma(s) \subset A \tag{2}$$

We can now take an example from Milner's text and represent it as deterministic state machine — albeit awkwardly. A cleaner formulation with string functions is in section 3. 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 D_2 may non-deterministically enable both b and c or only c.

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

In this case, we have two state machines D_1 and D_2 and we know that

$$\forall r \in R[\gamma_1(\delta(start_1, r) = \{a\} = \gamma_2(\delta_2(start_2, r))]$$

and

$$\forall r \in R[\gamma_1(\delta_1(\delta_2(start_1, r), a)) = \{b, c\}]$$

$$\forall r \in R[\gamma_2(\delta_2(\delta_2(start_1, r), a)) \in \{\{c\}, \{b, c\}\}]$$

2.2 Concurrency

Following Milner suppose there is a matching function on A that associates each $a \in A$ with a unique $\bar{a} \in A$. For convenience suppose $\bar{\bar{a}} = a$. The intuition is that a and $\bar{a}$ are opposite sides of a communication. When we compose two process algebra processes P_1 and P_2 to get some $P = (P_1|P_2)$ the intuition is that an input can either advance one of the two components or can advance both via matching inputs. The second case can only happen if the input applied to the composed process is the special input τ.

Suppose P_1 and P_2 are process algebra compatible and we want to construct $P_1|P_2$ as a state machine. In order to emulate composition we actually connect three state machines P_1, P_2 and some Z which acts as a scheduler. The state machine Z is not process algebra compatible but it makes the implicit scheduling that is part of process algebra composition into explicit scheduling. Let's step back an consider automata interconnection in a more general setting.

The "general product" connects a collection of state machines together via a "connection map". When an input a is applied to the product, the connection map computes a sequence (possibly a null sequence) of inputs for each component from a and from the outputs of some (or none or all) of the components. Suppose that $M_1 \dots M_n$ have been connected in this way. Then the states of the composite system are vectors $\vec{s} = (s_1 \dots s_n)$ where each $s_i \in S_i$ is a state from the state set of M_i. The output of this machine is a vector $\vec{x} = (x_1 \dots x_n)$ where each $x_i = \gamma_i(s_i)$ is the output produced by the output map γ_i of M_i when applied to the state element s_i. The components are connected by a connection map ϕ so that when a is applied to the product, each component M_i is advanced by the sequence of inputs $\phi(a, i, \vec{x})$ where $\vec{x}$ is the output of the current state. If we want the state of component i to remain unchanged, we can make $\phi(a, i, \vec{x}) = \epsilon$ where ϵ is the empty string.

If we connect M_1 and M_2 which are emulating process algebra processes and the respective outputs in the current state are x_1 and x_2, then an input a is enabled only if it is enabled by at least one of the two components or if $a = \tau$ and there is an enabled matching pair input. Define

$$Enabled(x_1, x_2) = \begin{cases} x_1 \cup x_2 \cup \{\tau\} & \exists b \in x_1 \text{ s.t. } \bar{b} \in x_2; \\ x_1 \cup x_2 & \text{otherwise.} \end{cases}$$

If a is enabled and applied to a product containing state machines P_1 and P_2 then:

- If $a \in R$ then $\phi(a, i, \vec{x}) = \langle r \rangle$ (just pass the random seed along).
- If $a \in A$ then either:
  - $a \in x_1$ and $\phi(a, 1, \vec{x}) = \langle a \rangle$ and $\phi(a, 2, \vec{x})) = \epsilon$.
  - OR $a \in x_2$ and $\phi(a, 2, \vec{x}) = \langle a \rangle$ and $\phi(a, 1, \vec{x})) = \epsilon$.
- If $a = \tau$ then for some $b \in A$, $\phi(a, 1, \vec{x}) = \langle b \rangle$ and $\phi(a, 2, \vec{x}) = \langle \bar{b} \rangle$ and $b \in x_1$ and $\bar{b} \in x_2$.

And that's pretty much it — except for what to do if more than one of these options is true or if there are more than a single pair of matching enabled transitions. We could put the choice in ϕ, but that would violate the spirit of the process algebra composition — which may allow different choices in the same configuration. That's why we need Z which has totally unspecified operation but introduces an additional state component that may modify how ϕ operates. To make this all precise in a more compact format, I'll turn to string functions.

3 String functions

Each state machine M determines a map from strings to outputs that is more convenient for our purposes than the tuple-of-sets-and-functions form. Let A^* be the set of finite strings over A including the empty string ϵ and if $w \in A^*$ then let $wa \in A^*$ be the string obtained by appending a to w. Then extend δ to strings by $\delta(s, \epsilon) = s$ and $\delta(s, wa) = \delta(\delta(s, w), a))$. Mildly abusing notation, let $M(w) = \gamma(\delta(\mathit{start}, w))$. Then $M(w)$ is the output of M in the state reached by following w from the initial state.

Let $q \cdot w$ indicate that string w is to be concatenated onto the right side of string q.

Suppose $M_1, \ldots M_n$ are state machines. A product M is constructed from an appropriate connection map ϕ by:

$$M(w) = (M_1(w_1(w)) \ldots, M_n(w_n(w))) \tag{3}$$

$$\text{where } w_i(\epsilon) = \epsilon \tag{4}$$

$$\text{and } w_i(wa) = w_i(w) \cdot \phi(i, a, M(w)) \tag{5}$$

It can be shown that this definition corresponds to the general product, that if all of the factors are finite the product will also be finite. A more detailed exploration of this method can be found in [Yod09].

Let rw indicate the string obtained by appending r to string w on the left. Compare the following to definition 2.1.

Definition 3.1 *A state machine M is pseudo-random process algebra compatible for random input set R and input alphabet A iff*

$$\forall r \in R, M(\epsilon) = R \tag{6}$$
$$\forall r \in R, w \in A^*, M(rw) \subset A \tag{7}$$

Definition 3.2 *A product $M = (M_1(w_1(w)), M_2(w_2(w)), Z(w))$ with alphabet A and random seeds R is process algebra composition compatible iff*

$$M_1 \text{ and } M_2 \text{ are process algebra compatible} \tag{8}$$
$$\text{and for } x_i = M_i(w_i(w)), w_i(w) = q_i \text{ and } w_i(wa) = q_i \cdot z_i \tag{9}$$
$$\text{If } a \in Enabled(x_1, x_2) \text{ then}$$
$$a \in R \text{ and } z_i = \langle r \rangle \tag{10}$$
$$\text{or } a \in x_1 \text{ and } z_1 = \langle a \rangle \text{ and } z_2 = \epsilon \tag{11}$$
$$\text{or } a \in x_2 \text{ and } z_2 = \langle a \rangle \text{ and } z_1 = \epsilon \tag{12}$$
$$\text{or } a = \tau \text{ and } z_1 = \langle b \rangle \text{ and } z_2 = \langle \bar{b} \rangle \text{ for } b \in\in x_1 \text{ and } \bar{b} \in x_2 \tag{13}$$

3.1 Constructors and proofs

The constructors of process algebra can now be defined as operations on process algebra compatible state machines.

- Given P, define $P' = a.P$ by $P'(r) = \{a\}$ and $P'(raw) = P(rw)$.
- The "restriction" requires primitive recursion on strings:

$$(P \setminus H)(\epsilon) = P(\epsilon) \setminus H \tag{14}$$
$$(P \setminus H)(wa) = \begin{cases} P(wa) \setminus H & \text{if } a \in (P \setminus H)(w) \\ \text{not defined} & \text{otherwise} \end{cases} \tag{15}$$

- Relabeling is straightforward:

$$(P[f])(w) = P(f^*w) \text{ where } f^*\epsilon = \epsilon, f^*(wa) = (f^*w)f(a)$$

- Summation is a constraint with many potential solutions. Given P_1 and P_2, say Q is a solution to $(P_1 + P_2)$ if and only if each random seed "chooses" one P_i and each P_i is chosen by at least one random seed:

$$\forall r \in R, \exists i \in \{1,2\} \text{ s.t. } \forall w \in A^*Q(rw) = P_i(rw) \quad (16)$$

$$\text{and } \forall i \in \{1,2\} \exists r \in R \text{ s.t. } \forall w \in A^*Q(rw) = P_i(rw) \quad (17)$$

- Composition is easy. Given P_1 and P_2, say Q is a solution to $(P_1|P_2)$ if:

$$Q(w) = Enabled(P_1(w_1(w)), P_2(w_2(w))) \quad (18)$$

$$\text{where } M(w) = (P_1(w_1(w)), P_2(w_2(w)), Z(w))$$

$$\text{is a process algebra compatible product}$$

$$\text{for some state machine } Z \quad (19)$$

Say rw is a permitted behavior of P if every input was enabled.

$$Permitted(r, P) = TRUE, \quad (20)$$

$$Permitted(rwa, P) = \begin{cases} TRUE & \text{if } Permitted(rw, P) \text{ and } a \in P(rw); \\ FALSE & \text{otherwise.} \end{cases} \quad (21)$$

The accepted language of P is then $L(P) = \{rw : r \in R, w \in A^*, Permitted(rw, P)\}$.

The process algebra assertion $P \xrightarrow{a} P'$ means that P can make an a transition and then it will behave just like P'. More formally, $\forall rw \in L(P')P(raw) = P'(rw)$ The "rules" of Process algebra are then just assertions about permitted strings.

- $a.P \xrightarrow{a} P$ means $\forall rw \in L(P), (a.P)(raw) = P(rw)$.

- Suppose $P_i \xrightarrow{a} P_i'$, then let Q be a solution to $P_1 + P_2$, and it follows immediately that for at least one $r, \forall rw \in L(P_i), Q(rw) = P_i(rw)$ and since $au \in A^*$, $Q(rau) = P_i'(ru)$. so let $w = au$ to get $Q(rau) = P_i(rau) = P_i'(rw)$.

- Suppose $P_1 \xrightarrow{a} P_1'$, then let Q be a solution to $P_1|P_2$. We want to prove that $Q \xrightarrow{a} Q'$ where Q' is a solution to $(P_1'|P_2)$. But this follows from the construction.

4 Critique

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

From the point of view of automata theory, the "rules" of process algebra appear weak. That is, tools for showing that a single initial step from process P causes future steps to appear as if they were from process P' seems hardly adequate for illuminating the behavior of systems that take many steps. The model of concurrency seems much too dependent on an artifact of early "threaded" programming languages and communication as synchronous exchange is limiting.

The exercise also shows 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 type of concurrency corresponds to a restriction of parallel products with a limited type of feedback.

Another issue is non-determinism. Non-determinism is a peculiar property to consider fundamental given the effort put into engineering computing devices to be deterministic. Even non-determinism in threading can be considered harmful[MOA09, Lee06]. As a tool for avoiding complexity in specifications, non-determinism may be justifiable on pragmatic grounds, given positive results. But one would have a hard time showing that process algebra has met such a test. In fact, the initial decision to drop "classical automata" as a basis is worth a revisit. "Classical automata" do not share the lack of clarity that Hennessy noted in the citation above and they have a strong mathematical connection to the study of semigroups[Arb69, Hol83]. Given that communication and interaction even in the rather complex approach taken in the process algebras can, as demonstrated above, be expressed completely within the domain of deterministic automata, the decision to create a new and more syntactic model seems poorly motivated.

References

[Arb69] Michael A. Arbib. *Theories of Abstract Automata*. Prentice-Hall, 1969.

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

[DN04] Pal Domosi and Chrystopher L. Nehaniv. *Algebraic Theory of Automata Networks (SIAM Monographs on Discrete Mathematics and Applications, 11)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2004.

[Gec86] Ferenc Gecseg. *Products of Automata*. Monographs in Theoretical Computer Science. Springer Verlag, 1986.

[Hol83] W.M.L. Holcombe. *Algebraic Automata Theory*. Cambridge University Press, 1983.

[HS66] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, 1966.

[Lee06] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[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.

[MOA09] Jason Ansel Marek Olszewski and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of ASPLOS'09: The International Conference on Architectural Support for Programming Languages and Operating Systems, Washington DC, USA*, March 2009.

[Yod09] Victor Yodaiken. Primitive recursive presentations of automata and their products. *CoRR*, abs/0907.4169, 2009.