PFPL Supplement: Automata and Concurrency*

Robert Harper

Fall, 2019

1 Introduction

Concurrency in programming may be understood as a generalization of classical automata theory to account for concurrent interaction among multiple automata. Classical automata theory makes no provision for defining automata by composition of others. Rather, they are abstract machines given in their entirety that act on separately-given data, thought of as reading from, or writing to, an input, or output, "tape." This note considers a minimalist language for concurrent programs with no separation between control and data, and with a compositional notation for constructing them. It is thus a *linguistic* alternative to the *mechanistic* classical formulation. Moreover, it provides a bridge to the process calculus considered in **PFPL**.

The correspondence between automata and concurrency is implicit in Milner's investigations of process calculus, summarized in his monograph Milner (1999).

2 Classical Automata Theory

Classically, an automaton, A, is given by specifying three pieces of information:

1. Its set $|A|$ of *states*, usually required to be finite.
2. Its *alphabet*, Σ_A, of *symbols*.
3. Its *transition relation* $\underset{A}{\mapsto} \subseteq |A| \times \Sigma_A \times |A|$, together with a specification of its *initial* and *final* state.

The a-labeled transition from state s to state s' is written $s \overset{a}{\underset{A}{\mapsto}} s'$, and the unlabeled transition from s to s' is written $s \underset{A}{\mapsto} s'$. By convention all automata have a single initial state and a single final state; if necessary, unlabeled transitions may be used to comply with this requirement.

Automata are usually depicted as graphs with labeled and unlabeled edges, as illustrated in Figures 1, 2, and 3. The initial state is marked with an incoming arrow, and the final state is labeled with a check mark.

A *word* w over an alphabet Σ is an element of the *free commutative monoid* generated by Σ, with (the insertion of) $\epsilon \in \Sigma$ being the unit element, and the multiplication written $w_1 \cdot w_2$. The symbol $a \in \Sigma$ is identified with the length-1 word given by the insertion of the generators.

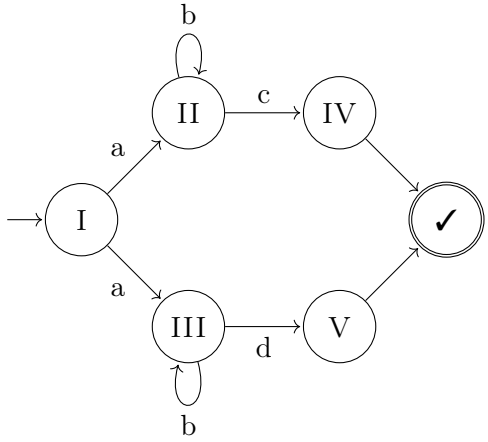

Figure 1: Automaton A_1

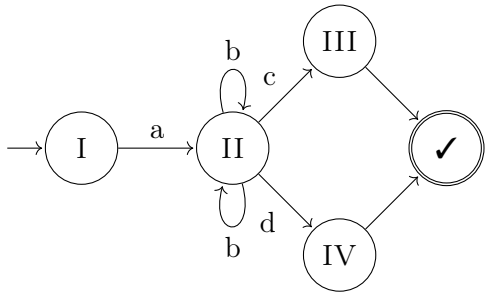

Figure 2: Automaton A_2

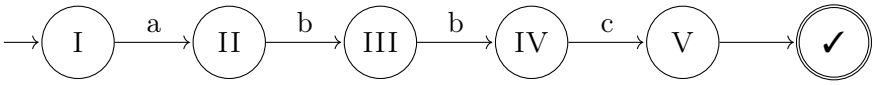

Figure 3: Automaton S

Define *multi-step labeled transition* $s \xmapsto[A]{w} s'$, which extends labelled transition for A to words, as follows:

1. $s \xmapsto[A]{\varepsilon} s$;
2. if $s \xmapsto[A]{w} s' \xmapsto[A]{w'} s''$, then $s \xmapsto[A]{w \cdot w'} s''$.

Notice that unlabeled transitions are absorbed by the monoid structure on words.

It is sometimes helpful to consider that if $s_0 \xmapsto[A]{w} s'$ is a transition from the initial state of A, then A has been transformed into an automaton A' with the same transitions as A, but with initial state s instead of s_0. Thus, by considering the start state as the "name" of an automaton, state transition can instead be viewed as transitioning from one automaton to another.

The *language* of an automaton A, written $\mathcal{L}(A)$, is the set of words w such that $s_0 \xmapsto[A]{w} \checkmark$, where s_0 is the initial state and 3 is the final state. That is, the words in the language are the concatenations of the labels on edges forming a path from the initial to the final state. An automaton is said to *accept* the words $w \in \mathcal{L}(A)$, and is also said to *generate* the same words.[1] It is easy to check that $\mathcal{L}(A_1) = \mathcal{L}(A_2)$, because they accept/generate the same words. In fact, their common language is given by the regular expression

$$\mathcal{L}(ab^*(c+d)) = \{\, ab^{(n)}c \mid n \geq 0 \,\} \cup \{\, ab^{(n)}d \mid n \geq 0 \,\}.$$

It also easy to see that $\mathcal{L}(S) = \{\, abbc \,\}$, a single string accepted by both A_1 and A_2. (The relevance of this example will become clear shortly.)

In automata theory it is customary to say that two automata, A and B, are *equivalent* iff they accept/generate the same language: $\mathcal{L}(A) = \mathcal{L}(B)$. This is a reasonable definition insofar as the only interest of an automaton is the language it accepts/generates. But when viewed as the *means* for achieving acceptance/generation (that is, when viewed as *programs*) this notion of equivalence is too coarse. Instead, when considering them as interacting agents, two automata are deemed equivalent iff they exhibit the same patterns of interaction. More precisely, two automata, A and B, are considered equivalent whenever one can make a transition to A', then B can make the same transition to B' with A' and B' being again equivalent in this sense.[2]

The automata A_1 and A_2 given in the figures are *not* equivalent when viewed as programs. From their start state both can both make an a-transition, but the states to which they transition are not then equivalent. Specifically, automaton A_1 makes an up-front commitment on accepting a to there being a subsequent c- or d-transition, whereas automaton A_2 is capable of either a c- or d-transition after accepting a. This is grounds for distinguishing them as programs, for reasons to be made clear in the next section.

3 Interacting Automata

The dual view of an automaton as being an acceptor or a generator is an elegant aspect of the theory, but the separation of an automaton from its input or output is decidedly not, at least

[1] It is a matter of preference whether to speak of acceptance or generation, according to whether the word is viewed as being read or written by the automaton.

[2] The "circular" nature of this description demands further justification, which is not given here, but see Chapter 49 of **PFPL** for a thorough discussion.

from a programming perspective. If automata are viewed merely as recognizers or generators, then perhaps this is alright, but when viewed as programs it is useful to be able to describe how the input is produced or output is consumed. This this achieved by shifting the duality from the "metalevel" of automata theory to the "object level", by introducing the concept of an *action* that is then used as the basis for defining *inter-action*.

Rather than label transitions with symbols from an alphabet, let us instead label them by *symbol-indexed actions* α defined as follows:

$$\alpha ::= a\,! \mid a\,? \mid \varepsilon$$

The silent action is denoted ε in agreement with the usage in the previous section. The other two forms of action may be considered as "asserting" or "signalling" or "writing" a symbol, and as "querying" or "sensing" or "reading" a symbol, respectively. But there is nothing inherent in those readings. What matters is that they are *complementary* in that $\overline{\alpha}$ is the complement, or dual, of α, according to the following definition:

$$\overline{\varepsilon} \triangleq \varepsilon \qquad\qquad \overline{a\,!} \triangleq a\,? \qquad\qquad \overline{a\,?} \triangleq a\,!$$

Thus, the ambiguity between acceptor and generator in automata theory transposes into the duality between actions here. The silent action is defined to be self-dual, as a matter of technical convenience.

The automata considered in the previous section may be reformulated using actions to make explicit, say, that A_1 is to be thought of as an acceptor, by labelling its transitions as query actions, and that S is, correspondingly, to be thought of as a generator, by labelling its transition as signal actions. (See Figure 4.) Thus, the "input" read by A_1 is written as "output" by S, giving pride of place to the "tape" within the language as a generator. The roles could, of course, be reversed, rendering A_1 as a generator, and S as an acceptor, without material difference. All that matters is that the actions are complementary, not the particular choice of polarity.

The purpose of this change is to consider the behavior of the *concurrent composition* of the two automata, illustrated by their juxtaposition in Figure 4. Now, rather than be driven by an extrinsic execution mechanism, the combined automata can interact according to the principle that if they *separately* can take a complementary transition to the other, then they can *jointly* take an unlabeled (ε-labeled) transition in which both advance to their respective next states. In fact by taking a series of such transitions the combination of A_1 and S (in their polarized forms) can jointly progress *unlabeled transitions* to their designated final states, which may be interpreted as automaton A_1 accepting the string generated by automaton S. Had S' been defined the same as S, but for the action $c\,!$ being replaced by $d\,!$, then A_1 and S' would similarly progress by silent actions to their final states. On the other hand, were $c\,!$ replaced by, say, $a\,!$, then no such transition sequence would be possible; the automaton A_1 rejects that input.

The process of acceptance just described involves *non-deterministic choice* in determining whether A_1 will, in response to the $a\,!$ action proceed to its state `I` or `II`. Nothing forces that choice. Rather, the property of interest is whether the combined automata *can*, by some series of choices, progress to their respective final states. This is an existentially quantified statement, and essentially so, for if A_1 were to choose the wrong transition, then it could not progress to completion in conjunction with S. Contrast the situation with a suitably prepared (with query actions) version of A_2 juxtaposed with either S or S'. Now *every* transition from the initial state (there is only one) proceeds to a state that can, at the right moment, choose whether to query c or d. In this case the choice is *causal*

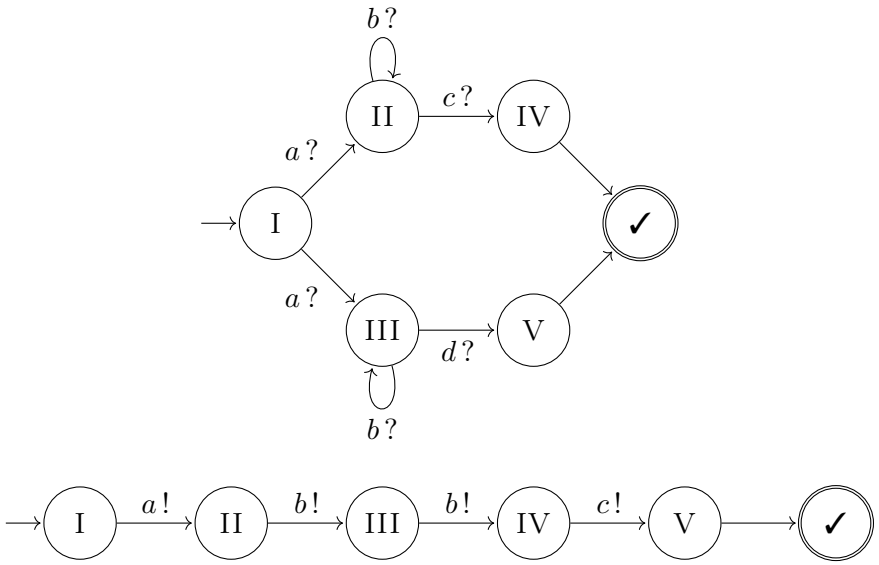

Figure 4: Communicating Automata

in that the available signal action from S determines whether to stand in place (in response to a b) or to progress to state `IV` or `V` in response to a c or d, respectively, and from there, silently, to the final state. Contrarily, A_1's choice is *non-causal*: it must peer into the future to "know" which way to go in response to an a signal. Making the wrong choice can cause A_1 to "get stuck" with no further interaction possible, whereas no such conundrum can arise with A_2. *It is for this reason that the two automata, A_1 and A_2 cannot be regarded as equivalent.*

The example just considered represents the input (or output) as a simple automaton with no choices in its evolution. That suffices for the representation of a single string, but nothing prevents us from considering, say, the pair of strings *abbc* and *abbd* as a single process that "forks" at the last stage to choose between c and d. (See Figure 5.) In conjunction with an acceptor the choice is causal, the willingness of the acceptor forcing one choice or the other. And there is no reason to insist that automata consistently play only the acceptor or generator role. They may, instead, "switch sides," as it were, as the computation proceeds. These possibilities are precluded by the simple read/write tape formulation of automata in the standard literature.

It should be stressed that composition cuts both ways in that the combination of several automata preserves the transitions of each of them separately, and all of the pairwise interactions that they may jointly undertake. For example, the composition of A_1, A_2, and S given above may evolve according to the interaction of, say, A_1 and S, or A_2 and S, but it may also evolve by *intermixing* interaction of each of the A_i's with S, possibly in such a way that neither of them can reach a final state! The more the possibilities, the harder it is to *ensure* that the composite achieves a particuar end, such as recognition of the string generated by S. The composite naturally *allows* the "good" interactions, but it *does not preclude* the "bad" ones. Essentially all of the difficulties of concurrent programming are contained in this simple remark.

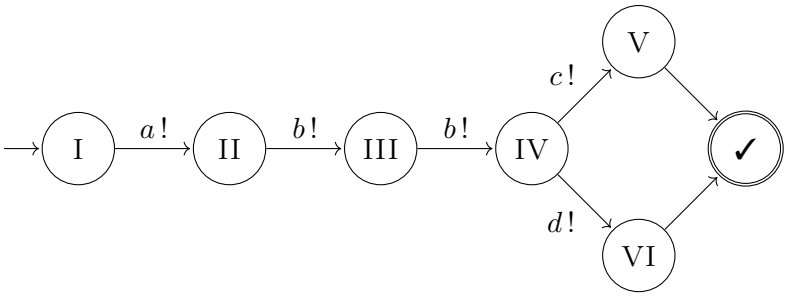

Figure 5: Non-Deterministic Generation

4 Compositional Formulation

It remains to define a proper programming language for interacting automata, a *calculus for concurrency*. As in **PFPL** the more-or-less standard notation is as follows. First off, "automata" are renamed "processes," to convey the idea of the in-progress execution of a program (and to connect with standard terminology in the systems literature.) The syntax of processes is given by the following grammar:[3]

$$P ::= \mathbf{1} \mid P_1 \otimes P_2 \mid \mathbf{0} \mid P_1 \oplus P_2 \mid !\, a\,;P \mid ?\, a\,;P \mid X \mid \mu X\,.\,P$$

The first two forms correspond to nullary and binary composition, the second two to nullary and binary choice, the third two induce signal and query actions, and the fourth two provide recursive definitions corresponding to circular graphs in the notation for automata. (The letter X is a process variable.)

Two judgments govern processes in this setting, *structural congruence*, written $P \equiv Q$, and transition, written $P \stackrel{\alpha}{\longmapsto} Q$. Structural congruence is defined to be the least equivalence relation satisfying the following conditions:

1. It is compatible with all process-forming constructs. So, for example, if $P \equiv Q$, then $!\, a\,;P \equiv !\, a\,;Q$, and so forth.

2. It specifies that $\mathbf{1}$ and $P \otimes Q$ form a commutative monoid. Thus, $P \otimes Q \equiv Q \otimes P$, and $\mathbf{1} \otimes Q \equiv Q$, and concurrent composition is associative.

3. It specifies that $\mathbf{0}$ and $P \oplus Q$ form a commutative monoid.

4. It defines that a recursive process is congruent to its unrolling, $\mu X\,.\,P \equiv [\mu X\,.\,P/X]P$.[4]

Decidedly, there is *no* distributive law relating concurrent composition and choice! By contrast in automata theory there would be such a relationship, because it is valid under the interpretation of automata as language acceptors/generators.

Labeled transition between processes is defined by the rules given in Figure 6. The CONG rule specifies that concurrently executing processes, and choice among processes, have no structure

[3]The syntax is parameterized by a fixed alphabet of symbols.

[4]It might be preferable to treat this not as a structural congruence, but by a rule of interaction that unrolls the recursion.

$$\text{CONG}\ \frac{P' \equiv P \qquad P \overset{\alpha}{\longmapsto} Q \qquad Q \equiv Q'}{P' \overset{a}{\longmapsto} Q'} \qquad \text{FRAME}\ \frac{P \overset{\alpha}{\longmapsto} P'}{P \otimes Q \overset{\alpha}{\longmapsto} P' \otimes Q} \qquad \text{INTERACT}\ \frac{P \overset{\alpha}{\longmapsto} P' \qquad Q \overset{\overline{\alpha}}{\longmapsto} Q'}{P \otimes Q \longmapsto P' \otimes Q'}$$

$$\text{CHOOSE}\ \frac{}{P \oplus Q \longmapsto P} \qquad \text{SIGNAL}\ \frac{}{!\,a\,;P \overset{a!}{\longmapsto} P} \qquad \text{QUERY}\ \frac{}{?\,a\,;P \overset{a?}{\longmapsto} P}$$

Figure 6: Process Transition

beyond the actions the individual components may take. The FRAME rule specifies that composition does not disrupt the actions of its components. The INTERACT rule defines the synchronization between two evolving automata induced by their ability to take complementary actions. The SIGNAL and QUERY rules define the actions that may be taken by a process, and thus correspond to the edges in its graph representation. Finally, CHOOSE makes a spontaneous (uncaused) choice between two processes, abandoning the other (in the particular line of execution starting with that choice.)

If recursive unrolling is not to be considered as a structural congruence, then the following rule should be added to those in Figure 6:

$$\text{MU}\ \frac{[\mu X\,.\,P/X]P \overset{\alpha}{\longmapsto} Q}{\mu X\,.\,P \overset{\alpha}{\longmapsto} Q}$$

Thus, the transitions from a recursively-defined process are just those of its unrolling.

Exercise 4.1. *Formulate automata A_1, A_2, and S as processes, and trace out their interactions, both in pairs and altogether.*

5 Controlling Interference

As soon as multiple interacting automata are considered it becomes difficult to ensure that the composition behaves as intended. For example, consider the concurrent composition $A_1 \otimes A_2 \otimes S$ of all three automata from the preceding section. Straight away, there is no means of ensuring that either A_1 or A_2 interact with S—either can synchronize with S and then continue the interaction it would have taken without the other being present. This may not seem problematic, but very similar-looking situations typify what can go wrong in concurrent programming.

Consider the composition $A \triangleq A_1 \otimes A_2 \otimes C \otimes S$, where $C \overset{c?}{\longmapsto} C'$ is an automaton accepting the symbol c. The composite automaton, A, may well start out with, say, A_1 interacting with S along a, and then, successively, twice more along b. All is well up to this point, but then C, chosen by the devil himself, can disrupt the interaction of A_1 with S by accepting c from S, thereby "stealing" it from A_1, which is then unable to reach completion. This phenomenon exemplifies the problem of

interference: the automaton C interferes with the "intended" behavior of the other three automata.[5]

How can such unintended interactions be avoided? Fundamentally, the issue is that C knows about the pathways for interaction between A_1 and S, which allows it to be chosen so as to interfere with their intended interaction. To guard against this behavior, it suffices to segregate the other processes from C by limiting the scope of the symbols a, b, and c, along which they interact, to themselves, excluding C. In process notation the process $\nu\, a\, .\, P$ restricts the symbol a to the process P by binding it with an abstractor. By the α-equivalence principle the symbol a becomes an unguessable secret within P, inaccessible to C. Formally, the restriction process obeys the following transition rule, which excludes actions that involve the private symbol a:

$$\frac{P \stackrel{\alpha}{\longmapsto} P' \qquad a \notin \alpha}{\nu\, a\, .\, P \stackrel{\alpha}{\longmapsto} \nu\, a\, .\, P'}$$

Returning to the example, the composite process

$$\nu\, a, b, c\, .\, (\, A_1 \otimes A_2 \otimes S\,) \otimes C$$

may only evolve as if C were not present; interference by C is impossible.

6 Summary

Classical automata theory should perhaps have been called *automaton* (singular) *theory*, the emphasis being on one automaton in isolation, activated extrinsically via an input or output "tape." Concurrency theory might, by the same token, have been called *automata* (plural) *theory*, because it studies the interaction of several automata, animated by the synchronization of complementary actions. In this setting there is no need to restrict to a machine model in which control is segregated from data, rather it is more natural to consider an integrated view of control and data based on interaction. Besides its intrinsic merits, the present theory of interacting automata leads directly to the richer concepts of concurrency discussed in **PFPL**.

References

Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, Cambridge, England, Second edition, 2016.

Robin Milner. *Communicating and Mobile Systems: The π-calculus.* Cambridge University Press, New York, NY, USA, 1999. ISBN 0-521-65869-1.

[5]The scare quotes signal that the intent is in the mind of the programmer, and is not inherent in the automata themselves.