An automata-theoretic approach to concurrency through distributive categories: on morphisms

N. Sabadini, S. Vigna and R.F.C. Walters

Dipartimento di Scienze dell'Informazione, Università di Milano, Via Comelico 39/41, Milan, Italy

FAX: +39-2-55006276; e_mail: sabadini@imiucca.csi.unimi.it

School of Mathematics and Statistics, University of Sydney, N.S.W. 2006, Australia

January 1993

In this paper we discuss some features of a calculus for automata based on distributive categories introduced in [SW93]. The class of automata we obtain provides a very clear environment in which to discuss issues arising from concurrency, asynchrony and parallelism. We give some examples of application of the calculus, and then we introduce a state-dependent notion of morphism. This notion seems to be general enough to provide a simple description of many interesting issues; for example, abstraction and refinement of automata, timing and input/output.

1. Introduction

In this paper we introduce a new notion of morphism of automata in the context of the *distributive automata*[1], an automata-theoretic approach to concurrency introduced by Sabadini and Walters in [SW93].

This model of concurrency follows the classical style of automata theory and formal language theory where the main issue is the relation between automata and their behaviour. More precisely, it is concerned with a *calculus* for automata that produces new automata out of old, and how the behaviour of the new automata relates to the behaviours of the old. We observe that there is no simple general correspondence between the operations on processors and operations on processes which might allow one to ignore processors in favour of processes. Other theories such as the process algebras [Hoa85, Mil89, BK85], which are observationally oriented, are essentially based on operation on behaviours (i.e., processes), but they lack an analysis of the corresponding constructions on machines; in fact, the *operational semantics* for an algebraic calculus like CCS is a nondeterministic automaton (non necessarily finite) which is canonically built from a term of the language: then, it is not usually the case that operators of the language induce basic construction on automata (see, for instance, the parallel composition operator). Automata oriented theories, like UNITY and asynchoronous automata, are much closer to our approach but lack its compositionality and hierarchical modularity. Intermediate approaches are trace theory [Maz77] and Petri nets [Rei85].

[1] Distributive=distributed+iterative.

Of course, on the firm foundation of state and behaviour, abstractions are necessary to analyse state and behaviour.

What is particularly novel in this approach is the *method of construction* of automata considered; we have a two-sorted theory, with operations between the sorts, the two sorts being: (1) deterministic automata on finite alphabets, and (2) functions. The functions are to be thought of as operations on data types. The basic idea is to take some automata and some functions (i.e., data types) and produce new automata and functions using the operations of a *distributive category*, that is, by using *sums* and *products* and the law of distribution of products over sums.

We observe also that an automaton with state space of the form $X + U + Y$ may compute by iteration a function from X to Y. We call such an automaton a *functional processor* or *pseudofunction*. The *replacement* of the automaton by the function it computes is the last operation of our calculus. It combines iteration and hiding of state and time. Since this operation is used to product *atomic* actions of new automata, we require that the function computed be *total*. This allows one to define *library calls* which can be subsequently used in the language. A precursor to the notion of pseudofunction is contained in [Elg75], and is the basis of iterative theories. A similar notion also occurs in [Hel90], and in UNITY [CM88].

Automata constructed in the above way from basic data types seem to be the appropriate ones for the consideration of concurrency and distributed processing; the necessity for iteration is clear; the operations of sum and product are essential for any such consideration since they express, respectively, total conflict and independent parallelism.

One of the purposes of this paper is to discuss in more detail an interesting feature of the model, namely that also the *alphabet* of an automaton may be constructed using sums and products of sets and hence may have a rich structure. For example, the intended meaning of $A_1 \times A_2$ is that actions are pairs of symbols acting *simultaneously*.

The distinction between structure on the alphabet and structure on the state space, and relations between them, give us a clear framework in which various concept of concurrency can be discussed. For example, the distinction between *independence* of actions and *parallelism* of their execution [KMS91].

In particular, even if our machines are deterministic, we can describe both synchronous and asynchronous devices by exploiting the rich structure of the alphabet. In fact, our machines generalize the asynchronous automata of Zielonka [Zie85]: the state space of this model of automata is given by a product and, in the description given by Zielonka, the alphabet is not structured. In our view, it is valuable to consider a more accurate description of the alphabet as a sum of products which allows us to express also the simultaneous occurence of actions in a trace. As a further example, we describe in detail a model of two machines connected by an asynchronous bus.

The other purpose of the paper is to introduce a new notion of morphism of automata, appropriate to this context. A special case of this notion was introduced in [SVW93] in order to study *refinement* of automata. A crucial feature we needed there to express questions of efficiency of resources and time was that the refinement of an action may be state dependent. This state dependence allows us also to consider other interesting phenomena, as the *timing* of actions, input/output (clearly output is dependent on both action *and* state), and abstraction.

The mathematical background needed for this paper is an elementary knowledge of category theory up to products and sums, and the distributive law between products and sums, such as contained, for example, in the first three chapters of Walters [Wal]. Other useful references for the categorical constructions on functions used in this paper are [Mac71].

We would like to thank Felice Cardone, Stefano Kasangian and Giancarlo Mauri for useful discussions. This work has been supported by the Australian Research Council, Esprit BRA ASMICS, Italian MURST 40%.

2. The model

In this section we briefly present our model. First of all, we introduce the operations we will use for producing new functions out of old. The notation used should be clear, but for further details see Walters [Wal].

Given a family of sets X, Y, $Z, \ldots$, (call them the *basic* sets), then consider sets which can be obtained from the empty set $\emptyset$, the set $I = \{*\}$ and the basic ones by repeated use of sum and product (call them the *derived* sets). Now suppose we have a family G of (*basic*) functions between derived sets. Any of the operations that we will introduce produces from these functions a new function between derived sets. The functions so obtained by repeated use of these operations, beginning with G, we call the *derived* functions.

In everything that follows there will be a given, usually unspecified, collection of basic sets and all functions will be between sets derived from these given ones.

The category of sets and functions is a *distributive* category [Wal]; that is it has finite products and sums (=coproducts=disjoints sums) and the canonical function $X \times Y + X \times Z \longrightarrow X \times (Y + Z)$ is an isomorphism. In such a category there are the following special arrows: the projection maps $proj_1 : X \times Y \longrightarrow X$, $proj_2 : X \times Y \longrightarrow Y$; the injections $inj_1 : X \longrightarrow X + Y$, $inj_2 : Y \longrightarrow X + Y$; the *diagonal* and the *codiagonal* functions $\Delta : X \longrightarrow X \times X$, $\nabla : X + X \longrightarrow X$; Further, given two functions $f : X_1 \longrightarrow Y_1$, $g : X_2 \longrightarrow Y_2$, there are the functions $f+g : X_1+X_2 \longrightarrow Y_1+Y_2$, $f \times g : X_1 \times X_2 \longrightarrow Y_1 \times Y_2$. Of course the composition of arrows are arrows.

If $f : X \longrightarrow Y$ and $g : X \longrightarrow Z$ and $h : Z \longrightarrow Y$ are functions then we denote the composite $\nabla \circ (f + h) : X + Z \longrightarrow Y + Y \longrightarrow Y$ by $(f|h) : X + Z \longrightarrow Y$; and we denote the composite $(f \times g) \circ \Delta : X \longrightarrow X \times X \longrightarrow Y \times Z$ by $(f, g) : X \longrightarrow Y \times Z$. Some other arrows it is useful to have names for: $twist : X \times Y \longrightarrow Y \times X$ for $(proj_2, proj_1)$ and $twist : X + Y \longrightarrow Y + X$ for $(inj_2|inj_1)$.

In this paper we use the words automaton and processor synonymously.

Definition 2.1. A *processor*, $\mathbf{X}$, with alphabet A, is defined by a set X, the *state space of* $\mathbf{X}$, and a function $\alpha : A \times X \longrightarrow X$. Then if $a \in A$, the function $\alpha_a : X \longrightarrow X$ which takes $x \in X$ to $\alpha(a, x)$ is called the *action* of the letter a on the state space.

A *process* of the processor is an infinite sequence of letters $a_0, a_1, a_2, \ldots$ and states,

$$x_0 \mapsto x_1 \mapsto x_2 \ldots \mapsto x_n \mapsto \ldots$$

such that, for each n, $x_{n+1} = \alpha_{a_n}(x_n)$.

As we described in the introduction, the novelty of our approach is that both the state space X and the alphabet A of a processor are *structured sets*, built in a hierachical way from the basic sets using the operations of sum and product described above.

An important special case is when the alphabet A consists of one element. In this case the process consists of a state space X and a single loop $\alpha : X \longrightarrow X$.
In the following if no alphabet is mentioned in referring to a process it is intended that the alphabet has one letter which we do not name.
It is clear, as we remarked above, that since the actions of processors are themselves special functions between derived sets they can be constructed using operations on functions.

2.1. *Simple operations on processors*

We give immediately some very basic examples of operations on processors. Note that the operation described work on *two* levels, i.e., state (implementation) and actions (specification), which interact. This allows us to formalize in a very simple way phenomena such as the synchronous and asynchronous execution of actions, or conflicts.

2.1.1. *Parallel product of processors* Given two processors $\mathbf{X}$, $\mathbf{Y}$, with alphabets A_1, A_2, state spaces X, Y, actions α, β, respectively, we can form a new processor $\mathbf{X} \times \mathbf{Y}$ with alphabet $A_1 + A_2$ as follows: state space $X \times Y$, and action $\gamma_a = \alpha_a \times 1_Y : X \times Y \longrightarrow X \times Y$ if $a \in A_1$, and $\gamma_b = 1_X \times \beta_b : X \times Y \longrightarrow X \times Y$ if $b \in A_2$.

It is clear that the processes of this new processor consist of *interleaved* processes of the two processors.

$A_1 + A_2$ is not the only alphabet which acts sensibly on $X \times Y$. Another choice is $A_1 \times A_2$ with action $\delta_{(a,b)} = \alpha \times \beta$. Processes of this processor are synchronized parallel processes of $\mathbf{X}$ and $\mathbf{Y}$.

Another choice is $A_1 \times A_2 + (A_1 + A_2)$ (the *asynchronous alphabet*) with the actions obtained from the last two examples. The new processor might be thought of as an *asynchronous parallel product*. Trace theory [Maz77] is concerned with the fact that the action of $(A_1 \times A_2 + (A_1 + A_2))^*$ on $X \times Y$ factors through a free partially commutative monoid on $A_1 + A_2$.

Processes which act in parallel and communicate will be discussed in later sections.

2.1.2. *Totally conflicting processors* Again we could consider processors with alphabets as in the last example but for simplicity we restrict this example to one-letter alphabets.

Given two processors $\mathbf{X}$, $\mathbf{Y}$, with actions α, β, respectively, we can form a new processor $\mathbf{X} + \mathbf{Y}$ as follows: state space $X + Y$, and action $\alpha + \beta : X + Y \longrightarrow X + Y$. If the processor is initially in a state of $\mathbf{X}$ ($\mathbf{Y}$) then it will never exit from $\mathbf{X}$ ($\mathbf{Y}$, respectively). It is clear that the behaviours of this new processor is either a process of the first or a process of the second processor.

2.1.3. *Information hiding—pseudofunctions* Some processors (with single letter alphabets) are intended to compute functions by iteration.

This notion was made precise in Khalil-Walters [KW93], where the following definition of pseudofunction was introduced.

Definition 2.2. A *functional processor* or *pseudofunction* from X to Y is a processor with state space of the form $X + U + Y$, and action α such that

(i) for y in Y, $\alpha(y) = y$;
(ii) for each x in X there exists a natural number n_x such that $\alpha^{n_x}(x) \in Y$.

A pseudofunction $\alpha : X + U + Y \longrightarrow X + U + Y$ defines a function $\mathbf{call}(\alpha) : X \longrightarrow Y$ (by iteration) as follows:

$$\mathbf{call}(\alpha)(x) = \alpha^{n_x}(x).$$

We often denote the pseudofunction α as $\alpha : X \dashrightarrow Y$ to indicate that it computes a function from X to Y. We call U the set of *local* states of the pseudofunction α, or the *hidden* states of the function $\mathbf{call}(\alpha)$.

Note that condition (ii) is a semantic condition clearly undecidable in general. It is our view that when hiding state there is an obligation on the programmer to ensure that this semantic condition is satisfied.

3. Asynchrony

In this section we show how two asynchronous devices can be described in our formalism.

3.1. *Asynchronous automata*

One of the most widely studied automata-theoretic models of concurrency are the asychronous automata of Zielonka [Zie85], which were introduced as recognizers of trace languages.

Definition 3.1. An asynchronous automaton has a finite state space of the form

$$X = X_1 \times X_2 \times \cdots \times X_n$$

(i.e., n processors). Corresponding to each subset of the of the processors $U \subseteq \{1, 2, \ldots, n\}$ there is an alphabet A_U of those letters which are associated with the processors in the subset. Then, for each $a \in A_U$ there is a library function

$$f_a : \prod_{j \in U} X_j \longrightarrow \prod_{j \in U} X_j$$

The action of a can then be defined on the whole state space by multiplying it by the identity function on $\prod_{j \notin U} X_j$.

From our point of view the alphabet acting on the automaton is

$$A = \sum_{U \subseteq n} A_U + \sum_{\substack{U,V \subseteq n \\ U \cap V = \emptyset}} A_U \times A_V + \sum_{\substack{U,V,W \subseteq n \\ U \cap V = U \cap W = W \cap V = \emptyset}} A_U \times A_V \times A_W + \cdots$$

Notice that when $U \cap V = \emptyset$, if $a_u \in A_U$ and $a_v \in A_V$ then

$$a_v a_u \cdot x = (a_u, a_v) \cdot x = a_u a_v \cdot x.$$

We can equate input words which have the same effect because of this commutation. This gives a monoid morphism

$$A^* \longrightarrow A^* / \langle a_v a_u \cdot x = (a_u, a_v) \cdot x = a_u a_v \cdot x \rangle,$$

where the codomain is the *trace monoid* of the automaton. Elements of the trace monoid are called *traces*.

More generally, given an alphabet A we can consider an abstract indipendence relation aIb which is symmetric and irreflexive; then if we form

$$M = A^* / \langle ab = ba \text{ iff } aIb \rangle$$

(the trace monoid of the independence relation) you can consider finite automata with M acting. There is a very famous theorem by Zielonka, stating that the languages recognized by trace monoid automata are exactly the languages recognized by asyncronous automata with the same independence relation.

We have described Zielonka's asynchronous automata because they are a famous model of asynchrony and concurrency; however, our model has considerable advantages: first of all, asynchronous automata are clearly a special case of ours; and they are finite, while ours are finitely *generated*. Asynchronous automata have no data types, while ours are based on data types, and have a full programming language; asynchronous automata have traces acting, while ours have general polynomials (generalized traces). Finally, in our model we can restrict communication to a channel, whereas asynchronous automata do not have this capacity.

As a final note, we want to give the following example: suppose we have a distributive automaton with state space $X \times Y + U$, with alphabet $A_1 + A_2 + B$. Suppose A_1 and A_2 are really acting only on X and Y, respectively, when applied to the first summand of the state space, and that the letters in B move the state from one summand to the other one. Then, if the state x belongs to $X \times Y$ it is the case that

$$ba_1a_2 \cdot x = ba_2a_1 \cdot x$$

for all $a_1 \in A_1$, $a_2 \in A_2$, but it can definitely happen that

$$a_1a_2b \cdot x \neq a_2a_1b \cdot x,$$

because there is no guarantee that A_1 and A_2 have commuting actions on U. Thus, in this case, the language recognized by the automaton is not a trace language. A further analysis of what is really acting will be given in a forthcoming paper.

3.2. *An asynchronous communication protocol*

We now give a precise description in our model of an event-driven, two-phase protocol based on two communication request/acknowledgment lines, which was developed in order to design completely asynchronous chips (i.e., chips without an external clock) [Sut89]. We observe that to give a precise description should be the first step of a correctness proof for any distributed protocol. The communication take place between two subcomponents, a sender and a receiver.

Informally speaking, the handshaking works as follows: when the sender wants to send some data, it puts it on the data bus and changes the state of the request line; then it waits for a change of state of the acknowledgment line, and then it resumes its normal activites. When the receiver wants to receive some data, it looks at the state of the request line: if the state has changed from the last communication, it reads the bus and then changes the state of the acknowledgement line, otherwise it simply waits (always checking, of course, a change of state of the request line). Note that the protocol is driven by the *change of state* of the req/ack lines, and not by their state.

We consider *two one-letter processors* $\mathbf{X}$, $\mathbf{Y}$ whose state space is, respectively, $X_n + X_s$ and $Y_n + Y_r$ (we will name the respective time letters t_1, t_2). When either of the two processors is in the second part of its state space, it wants to respectively send or receive data; in order to describe the creation and usage of data, we have to ask for two arrows $p : X_s \longrightarrow X_s \times D$ and $c : Y_r \times D \longrightarrow Y_r$, the data-producing and data-consuming arrows.

The communication is handled by *two wires* between the processors, the communication request and communication acknowledgement lines. This part of the bus is represented by two sets $R = \{r_0, r_1\}$ and $A = \{a_0, a_1\}$, while the data part of the bus is represented by an object D which is left unspecified.

Finally, since our protocol is event-driven, every processor has to have an additional bit of memory which records the last state of the communication line controlled by the other processor. We call these additional bits B_R and B_A. Thus, $\mathbf{X}$ has now an enlarged state space given by $(X_n + X_s) \times B_A$, and analogously $\mathbf{Y}$ has an enlarged state space given by $(Y_n + Y_r) \times B_R$.

Thus, the state space of the whole system is given by

$$S = (X_n + X_s) \times B_A \times R \times D \times A \times B_R \times (Y_n + Y_r) = X \times C \times Y.$$

We suppose also the existence of arrows $inv_R : R \longrightarrow R$ and $inv_A : A \longrightarrow A$ which inverts the

state of the request and acknowledgment lines. Finally, there is an arrow $test_A : B_A \times A \longrightarrow B_A \times (\{t\} + \{f\})$ which tests for the state of A being changed from the information contained in B_A, updates B_A to the new state of A and set the third factor to true or false depending on the result of the test (analogously for $test_R$). Note that in what follows we will not explicitly write the permutation of factors or the projection on a part of a product whenever they are trivial.

We will now specify the behaviour of the system by breaking up the state space in a number of summands on which the action is more or less trivial. First of all, let us notice that the behaviour of $\mathbf{X}$ can change (apart from the internal state $X_n + X_s$) the state of the data bus D, of the memory bit B_A and of the request line R, but *not* of the rest of the state space. Analogously, the behaviour of $\mathbf{Y}$ can only change the state of the memory bit B_R and of the acknowledgement line A (which are the remaning factors). Thus, we will define an arrow

$$a : S \longrightarrow (X_n + X_s) \times B_A \times R \times D$$

and an arrow

$$b : S \longrightarrow A \times B_R \times (Y_n + Y_r)$$

which specify the behaviour of the first and of the second processor, respectively. Then the action of t_1, t_2 and (t_1, t_2) on S can be obtained as (a, b), $(a, proj)$ and $(proj, b)$ (where the projections have obvious domains and codomains).

We start by defininig a. First of all, we apply the projection

$$S \longrightarrow X_n \times B_A \times R \times D \times A,$$

which means that a does not depend on the state of $\mathbf{Y}$. Since the sum $X_n + X_s$ distributes, we can define the action a on the first summand by

$$X_n \times B_A \times R \times D \times A \longrightarrow X_n \times B_A \times R \times D$$

$$\stackrel{inj_1 \times 1}{\longrightarrow} (X_n + X_s) \times B_A \times R \times D \stackrel{t_1 \times 1}{\longrightarrow} (X_n + X_s) \times B_A \times R \times D$$

That is, while in normal state X does not look at the the communication bus, and just depends on its internal state.

We now define the action in the most interesting case:

$$X_s \times B_A \times R \times D \times A \longrightarrow X_s \times B_A \times A \times R \times D \stackrel{1 \times test_A \times 1}{\longrightarrow} X_s \times B_A \times (\{t\} + \{f\}) \times R \times D$$

We distribute then on $(\{t\} + \{f\})$. On the first summand, which represent a communication state in which the sender has data to send but the receiver has not yet acknowledged the previous communication, we simply apply the identity (and the obvious injection $X_s \longrightarrow X_n + X_s$). On the second summmand, instead, we apply

$$X_s \times B_A \times R \times D \longrightarrow X_s \times B_A \times R \stackrel{p \times 1 \times inv_R}{\longrightarrow} X_s \times D \times B_A \times R$$

$$\stackrel{(t_1 \circ inj_2) \times 1}{\longrightarrow} (X_n + X_s) \times D \times B_A \times R \longrightarrow (X_n + X_s) \times B_A \times R \times D$$

(note that you can get rid of the singletons by using canonical isomorphisms).

An analogous distribution on $Y_n + Y_r$ leaves to be defined the following part of the action b (the action on the first summand being always the identity):

$$R \times D \times A \times B_R \times Y_r \longrightarrow D \times A \times B_R \times R \times Y_r \stackrel{1 \times test_R \times 1}{\longrightarrow} D \times A \times B_R \times (\{t\} + \{f\}) \times Y_r$$

Again, after the distribution on $(\{t\} + \{f\})$, on the first summand we use the identity, while

on the second one

$$D \times A \times B_R \times Y_r \longrightarrow A \times B_R \times Y_r \times D \overset{inv_A \times 1 \times c}{\longrightarrow} A \times B_R \times Y_r \overset{1 \times (t_2 \circ inj_2)}{\longrightarrow} A \times B_R \times (Y_n + Y_r)$$

From the description is clear that the consuming and producing arrows can only be used when a state change of a communication line is detected. Not only: their use is always coupled to a change of state of the other line. This means that each read or write of the bus is *guarded* by an event and causes an event, which guarantees correctness. □

First of all, we want to notice that in our formalism it is immediately clear that the behaviour and the correctness of this protocol do not depend in any way on some kind of interleaving of the actions of the two components. Indeed, both actions can be applied at the same time without causing any harm.

Then, we want to point out that there are states in which the action of (t_1, t_2), $t_1 t_2$ and $t_2 t_1$ do not produce the same result. Thus, this protocol is beyond the descriptive power of asyncronous automata. Indeed, what this example show is that *parallel actions which may share the same resource, and hence are not independent, may not commute.*

Finally, note that we were able to completely describe the protocol without making *any* assumption on the nature of the communication request and acknowledgment lines, of the data bus and of the machine connected. Indeed, the event-driven nature of the protocol is completely hidden inside the specification of the arrows *inv* and *test*. Nothing prevents one to implement what we describe in a ternary rather than binary logic.

It could be objected that the description does not take into account that the communication lines are changes *after* some data is put on the bus; but clearly we could have easily added some extra state to achieve this result. In fact, the resulting system would be a *refinement* of the old one, in a precise sense that will be introduced in the next section.

3.3. *Atomicity*

Notice that in abstract automata there is no real notion of the timing of actions, and in fact the time taken by an atomic action may be state dependent. This models very well the situation in which external inputs are assumed to arrive only after the previous actions are completed. In this context the meaning of (a, b) acting is that *both* a and b are completed before another action takes place.

Internally to the model, this is represented by the fact that a computation may be made atomic and state hidden (Section 2.1.3) if there is a waiting loop at the end of the computation which allows the product of two new atomic actions to be synchronized.

In our view it an essential feature of atomic actions that they must terminate and when done in parallel must be synchronized. An important meaning of asynchrony is that observable behaviour does not depend on the timing of the atomic actions, even when atomic actions are encapsulated programs. Further aspects of timing are discussed in terms of our new morphisms of automata, in the last section of the paper.

4. Morphisms

In this section we discuss a new notion of morphism, which is a generalization of classical morphisms of automata consisting of a function on state which preserve the action (up to a fixed relabelling of the alphabet), and of morphisms consisting of a function on state which preserve the action up to the substitution of one letter with a fixed word in another alphabet. We will see that for our morphisms this substitution is state-dependent.

This notion seems to be general enough to provide a simple description of many intersting issues; for example, abstraction and refinement of automata, timing and input/output.

4.1. *Definition*

Definition 4.1. Suppose M is a monoid and $\mathbf{X}$ an M-automaton; that is, a set X together with an action of M on X, $M \times X \longrightarrow X : (m, x) \mapsto m \cdot x$; the action is required to satisfy the usual axioms $m_1 \cdot (m_2 \cdot x) = (m_1 m_2) \cdot x$ and $1 \cdot x = x$. Define the category $\mathit{Trans}(\mathbf{X})$ (the *transition system of* $\mathbf{X}$) as follows:

(i) objects are states (that is, elements) of X;

(ii) arrows from x to y are state transitions; that is, elements $m \in M$ such that $m \cdot x = y$;

(iii) composition is monoid multiplication.

A *mapping* $\mathbf{X} \longrightarrow \mathbf{Y}$ is a functor from $\mathit{Trans}(\mathbf{X})$ to $\mathit{Trans}(\mathbf{Y})$.

Notice that each arrow in $\mathit{Trans}(\mathbf{X})$ is determined by an element $m \in M$ *and* a domain and a codomain $x, y \in X$. Hence many distinct arrows will be labelled with the same element of the monoid.

If $M = A^*$ and $N = B^*$ then a mapping F from $\mathbf{X}$ to $\mathbf{Y}$ consists of two functions (which for sake of semplicity we call with the same name)

$$F : X \longrightarrow Y$$

and a function

$$F : X \times A \longrightarrow B^*$$

satisfying the condition that if $a \in A$ and $a : x \longrightarrow x'$ in $\mathit{Trans}(\mathbf{X})$ then $F(a) : F(x) \longrightarrow F(y)$ in $\mathit{Trans}(\mathbf{Y})$.

Proposition 4.1. If $\mathbf{X}$, $\mathbf{Y}$ are automata on free monoids A^*, B^*, and $\mathbf{X}$ is isomorphic to $\mathbf{Y}$ in the sense of definition 4.1, then $A \cong B$ and

$$F(a \cdot x) = \phi_x(a) \cdot (Fx),$$

where ϕ_x is a family of bijections $A \longrightarrow B$ indexed by X.

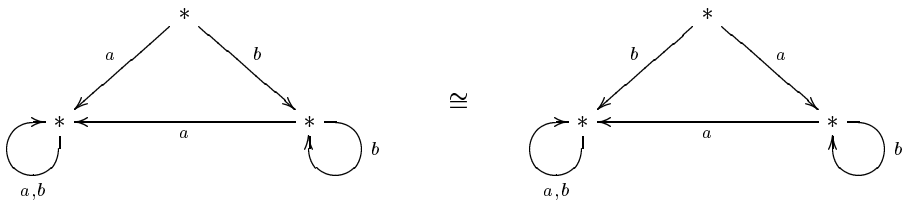
Proof. Suppose there is an isomorphism $F : \mathbf{X} \longrightarrow \mathbf{Y}$. Then F is a bijection on objects, i.e., a bijection between the sets X and Y, and a bijection between sets of arrows. Thus, every atomic arrow $x \xrightarrow{a} y$ has to be send to another atomic arrow $Fx \xrightarrow{\beta} Fy$, because the length of the image of an arrow can only increase both applying F and applying F^{-1}, so that

$$1 = |x \xrightarrow{a} y| \leq |F(x \xrightarrow{a} y)| \leq |F^{-1}F(x \xrightarrow{a} y) = x \xrightarrow{a} y| = 1.$$

Since this holds for F^{-1}, too, we can denote with ϕ_x the bijective restriction of F to the atomic arrows starting at x. Then ϕ_x is a bijection between A and B, because $\mathbf{X}$ and $\mathbf{Y}$ are deterministic. Moreover, $F(x \xrightarrow{a} y) = Fx \xrightarrow{\phi_x(a)} Fy$, which proves the last claim of the theorem. □

Note that we cannot claim that $\mathbf{X}$, $\mathbf{Y}$ are isomorphic as automata on A. A simple

counterexample follows:



If we want to fully recover the language-theoretical features of the automaton, we have to impose the condition $\phi_x = 1$, for all $x \in X$. In other word, we have to make the remapping state independent.

4.2. *Refinement*

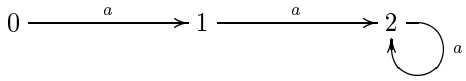
Our notion of morphism allows us to define a notion of refinement of automata that has been discussed in detail in [SVW93].

Definition 4.2. A mapping of automata $F : \mathbf{X} \longrightarrow \mathbf{Y}$ is a *refinement* iff the functor $F : \mathit{Trans}(\mathbf{X}) \longrightarrow \mathit{Trans}(\mathbf{Y})$ is a full inclusion.

In [SVW93] it is shown that in the case that $M = A^*$, $N = B^*$, if the automaton $\mathbf{Y}$ is a refinement of $\mathbf{X}$ then

(i) $Y = X + U$ for some set U;
(ii) for each $a \in A$, $x \in X$ there is an assigned word $b_1 \cdots b_n \in B^*$ such that $a{\cdot}x = b_1 \cdots b_n{\cdot}x$;
(iii) $b_1 \cdots b_k \cdot x \in U$ for $k = 1, 2, \ldots, n-1$.

Note that not *all* full inclusions in $\mathit{Trans}(\mathbf{Y})$ can have as domain a category of the form $\mathit{Trans}(\mathbf{X})$. For instance, consider the automaton



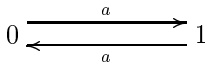
The full subcategory contaning the objects 0 and 1 has just one arrow $0 \xrightarrow{a} 1$; thus, it cannot be of the form $\mathit{Trans}(\mathbf{X})$ (not even if we consider the action of a general, non necessarily free, monoid M).

Note that if we consider the process backwards, the automaton $\mathbf{X}$ may be thought of as obtained from $\mathbf{Y}$ by a process of *information hiding*.

4.3. *Abstraction*

Definition 4.3. A mapping of automata $F : \mathbf{X} \longrightarrow \mathbf{Y}$ is an *abstraction* iff the functor $F : \mathit{Trans}(\mathbf{X}) \longrightarrow \mathit{Trans}(\mathbf{Y})$ is surjective on objects and arrows.

An abstraction is essentially a quotient of automata. Let us consider the following example: for any automaton $\mathbf{X}$, any partition of the states of $\mathbf{X}$ in two classes induces an abstraction F having as codomain the automaton $\mathbf{2}$ specified here:

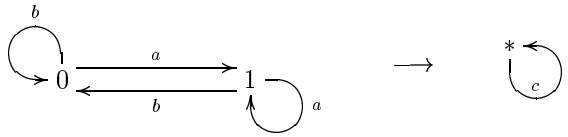


The abstraction maps the states in each class respectively to 0 and 1. All transitions which do not change the class are mapped to the identity of 0 or 1, respectively, while a transition which changes the class is mapped to the suitable a-transition of $\mathbf{2}$. As a result, each arrow

of $\mathit{Trans}(\mathbf{X})$ is mapped to an arrow of the form $x \xrightarrow{a^n} y$ (where $x, y = 0$ or 1); the exponent n is equal to the number of changes of class that happened while the state transformations induced by the arrow were applied.

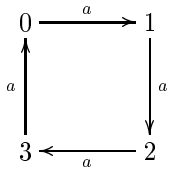
In other words, F forgets the information about the internals of the automaton, and allows us only to observe two states, and the transitions between those two states.

Note that *not all abstractions can be inverted by a refinement.* In other words, an epi $F : \mathit{Trans}(\mathbf{X}) \longrightarrow \mathit{Trans}(\mathbf{Y})$ does not necessarily have a section which is a full inclusion. The following counterexample shows this fact:



The mapping is defined by $a, b \mapsto c$. Any section of it has to set c to a or b. The inclusion cannot then be full because of the arrows $(ba)^n$ or $(ab)^n$, respectively.

The following example shows instead a case in which refinement of the abstraction is possible:



Consider a mapping from this automaton to **2**, defined by $n \mapsto n/2$ and by

$$0 \xrightarrow{a} 1 \;\mapsto\; 1_0,\; 1 \xrightarrow{a} 2 \;\mapsto\; 0 \xrightarrow{a} 1,\; 2 \xrightarrow{a} 3 \;\mapsto\; 1_1,\; 3 \xrightarrow{a} 0 \;\mapsto\; 1 \xrightarrow{a} 0.$$

The refinement $x \mapsto 2x$, with $x \xrightarrow{a} y \;\mapsto\; 2x \xrightarrow{a^2} 2y$ $(x, y = 0$ or $1)$, is a section of the given abstraction (the reader can check that there are four distinguished sections, which are all refinements).

4.4. *Input/Output*

In this section we note that our notion of morphism allows to specify output of automata. Our basic assumption is that output is *state-dependent*, i.e., it depends *both* on the state *and* on the letter of the alphabet which is acting.

Definition 4.4. An automaton on an alphabet A with output in the alphabet B is a A-automaton $\mathbf{X}$ endowed with a mapping $\mathbf{X} \longrightarrow B^*$ (where B^* is seen as a one-state automaton).

The meaning of this definition is that for each state x and each letter a of A there is a word w in B^* which represents the output produced by the action of a in the state x. Issues such as composition and the analysis of multiple I/O channels and connections will be addressed in a future paper.

4.5. *Timed machines*

We will consider machines with a *discrete timing*. This means we will assume the existence of a least (but variable) granularity of time, represented by N acting on a single state.

Definition 4.5. A timing for an automaton $\mathbf{X}$ is a mapping τ into the monoid N such the image of any atomic action is not zero. A timed automaton is an automaton with a specified timing.

Note that the timing is given in a *state-dependent way.*

We define the timing of pairs of letters in an asynchronous product as the *maximum timing* of the letters of the pair, following our assumption that it is characteristic of atomic actions that they must be synchronized.

Consider two timed machines with alphabet A and B, respectively, and consider an input word $(a_1, b_1) \cdots (a_n, b_n)$. The following trivial mathematical inequality:

$$\max(\sum \tau(a_k), \sum \tau(b_k)) \leq \sum \max(\tau(a_k), \tau(b_k))$$

expresses the fact that if we have a refinement of both machines to new machines whose atomic actions have length one, then the input word can be replaced in the asychronous product of the refined machines with a sequence of letters that produces the same state transformation in generally less time.

The follow proposition shows the existence of such a refinement for one-letter automata:

Proposition 4.2. Given a timed one-letter ($A = \{t\}$) automaton $\tau : \mathbf{X} \longrightarrow N$, there is a refinement $F : \mathbf{X} \longrightarrow \mathbf{X}'$ ($\mathbf{X}'$ being another one-letter automaton) such that if $x \xrightarrow{t} y \mapsto Fx \xrightarrow{s} Fy$ then $|s| = \tau(x \xrightarrow{t} y)$.

Proof. The refined state space is defined by

$$X' = \sum_{x \in X} \{0, 1, \ldots, \tau(x \xrightarrow{t} y) - 1\}.$$

The action on a state $\langle x, n \rangle$ is given by

$$\langle x, n \rangle \mapsto \langle x, n+1 \rangle \mapsto \cdots \mapsto \langle x, \tau(x \xrightarrow{t} y) - 1 \rangle \mapsto \langle y, 0 \rangle \mapsto \cdots.$$

Then, the inclusion defined by $x \mapsto \langle x, 0 \rangle$ induces trivially a refinement. □

References

[BK85] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[Elg75] C. Elgot. Monadic computation and iterative algebraic theories. *Studies in Logic and the Foundations of Mathematics*, 80:175–230, 1975.

[Hel90] A. Heller. An existence theorem for recursion categories. *Journal of Symbolic Logic*, 55(3):1252–1268, 1990.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, NJ, 1985.

[KMS91] S. Kasangian, G. Mauri, and N. Sabadini. Traces and pomsets: A categorical view. Technical Report 193, BRA Esprit DEMON, 1991. Final version in preparation.

[KW93] W. Khalil and R.F.C. Walters. An imperative language based on distributive categories II. *Informatique Théorique et Applications*, 27(6):503–522, 1993.

[Mac71] S. Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[Maz77] A. Mazurkiewicz. Concurrent program schemes and their interpretations. Technical Report DAIMA PB–78, Aarhus University, 1977.

[Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[Rei85] W. Reisig. *Petri Nets: An Introduction*. Springer-Verlag, 1985.

[Sut89] I. Sutherland. Micropipelines. *Communications of the ACM*, 32(6), 1989.

[SVW93] N. Sabadini, S. Vigna, and R.F.C. Walters. A notion of refinement for automata. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Proc. AMAST '93*, Workshops in Computing. Springer-Verlag, 1993.

[SW93] N. Sabadini and R.F.C. Walters. On functions and processors: an automata theoretic approach to concurrency through distributive categories. Mathematics Report 93-7, Sydney University, 1993. Available by anonymous `ftp` at `ghost.dsi.unimi.it` in the directory `pub2/papers/sabadini`.

[Wal] R.F.C. Walters. *Categories and Computer Science*. Carslaw Publications (1991), Cambridge University Press (1992).

[Zie85] W. Zielonka. Notes on finite asynchronous automata. *Informatique Théorique et Applications*, 27:99–135, 1985.