数理解析研究所講究録
960巻 1996年 15-22

Synchronization and nondeterminism

Gabriel Ciobanu*
Faculty of Computer Science
A.I.Cuza University
6600 Iasi, Romania
Gabriel@InfoIasi.Ro

Abstract

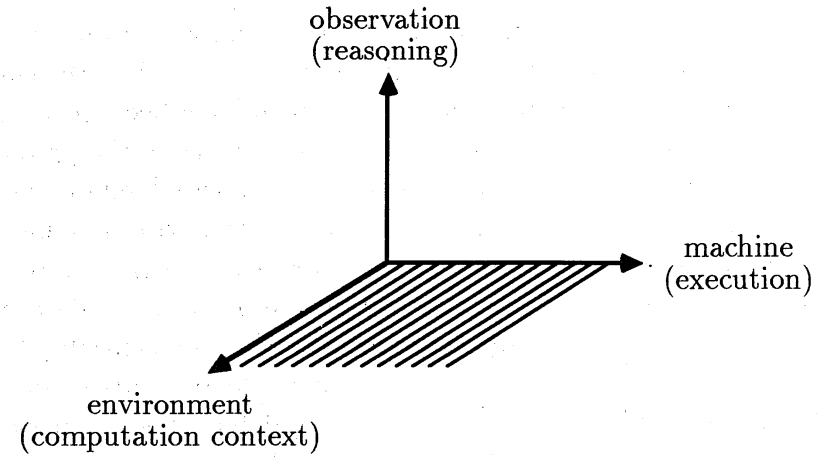
This paper describes a certain investigation of the nondeterministic phenomena discussing about internal and external, angelic and demonic forms of nondeterminism. We define and study algebraic structures of the set of actions together with the nondeterministic operations &, $\oplus$, and a synchronization operation $\|$. We use a dynamic logic construction to define a nondeterministic degree of each action, degree related also to a computational property. Some results connect this nondeterministic degree to the algebraic structures described in the first part of this paper.

1 Introduction.

Computer science considered "nondeterminism" as an interesting topic from very beginning (it would be enough to mention the wide interest in nondeterministic Turing machines). Various formalisms dealing with nondeterminism have been developed, to mention only a few examples: the powerdomains (Plotkin, Smyth), the predicate transformers for a choice construct (Dijkstra), and nondeterministic lambda-calculi. Models of concurrency reflect nondeterminism as nondeterminism arises in a natural way when dealing with concurrency. We would like to remark also the interesting viewpoint of Berry and Gonthier concerning this relationship between nondeterminism and concurrency, namely that "determinism does not mean sequentiality" and "most reactive systems can be decomposed into concurrent deterministic subsystems that cooperate in a deterministic way". Considering deterministic automata, Petri nets, CSP they say: "Quite amazingly, all the available techniques force the user to choose between determinism and concurrency, for they base concurrency on asynchronous implementation models where process non-deterministically compete

*Current address: Tohoku University, RIEC, 2-1-1 Katahira, Aoba-ku, Sendai 980, Japan.

for computing resources". This opinion reflects the complexity of the nondeterministic and concurrent computation. We believe that a better understanding of the (non)deterministic computation will give be a help in explaining the concurrent computation too.
Nondeterminism can appear also in programming languages, for example by nondeterministic assignment statement and nondeterministic do loop. The implementations of programming languages are usually deterministic, but the languages with an implicit nondeterminism (Prolog, for instance) allow more concise programs for many algorithms. Nondeterminism is used also as an abstraction concept in specification, whenever there is a hidden state or other components of a system which are conceptually, methodologically or technically inaccessible at a particular level of abstraction.
All these reasons are enough to justify our approach. Let define now our point of view. We would like to distinguish between the observation of the effects of a nondeterministic choice and the capability to perform an action in a nondeterministic way. Intuitively, the meaning of a nondeterministic choice between the actions A and B is given by: choose one of A and B, and perform the chosen action. This intuitive description gives rise to different interpretations (implementations). It is clear that we discuss about an observer, a computational environment and a machine. A machine performing the actions is strictly separated by an external observer at the run time, and both are working into a computational environment. The nondeterministic choice could be made by machine, by the external observer or by the computational environment. We can define a space of interaction by the following picture



and we work in, and refer to the space determined by the computational environment and the execution of a machine (program). For instance, we could imagine a printing program written in a nondeterministic programming language. We want to print a paper running this program. Nondeterminism may appear as a result of the programmer's (machine) choice. However the programmer has a deterministic mind

even when the execution is (apparently) nondeterministic. Some nondeterministic actions of a machine are determined by the other programs interfering with our printing program. Nondeterminism may also appear as a result of the computational environment. Even you action of printing a paper is clearly settled, you are not sure that this will happen: may be the printer is turned off, or it is turned on but it has no paper, or a linking cable is missing...
Finally, depending on who makes the choice, we consider two different forms of nondeterminism. The nondeterminism in which the nondeterministic choice is made by machine(program) is called internal; if the choice is made by the computational environment, then the nondeterminism is called external.
We refer in this paper to the executions of a machine (program). We may start from the T. Ito's Logic of Execution which was designed as a logical framework for executing sequences involving nondeterministic computation. In that logic processes are considered as sentences and $A \vdash B$ means that the success of execution of A implies(proceeds) the success of execution of B. An execution may become success $< s >$, failure $< f >$, or pending $< p >$. The set of "executional" operators contains the following operators:

- $A \oplus B$ for a "disjunctive execution"
  (i.e. "execute A and execute B, then lub(A, B)");
- $A\&B$ for a "conjunctive execution"
  (i.e. "execute A and B, then glb(A, B)");
- $A \| B$ for a "concurrent execution"
  (i.e. "interleaved execution of A and B").

Starting from the order $< f > \sqsubseteq < p > \sqsubseteq < s >$, Ito's executional operators satisfy the following properties:

$$\begin{array}{ll} A \sqsubseteq A \oplus B & B \sqsubseteq A \oplus B \\ A\&B \sqsubseteq A & A\&B \sqsubseteq B \end{array}$$

If $A \sqsubseteq B$ then

$$\begin{array}{ll} A \oplus C \sqsubseteq B \oplus C & C \oplus A \sqsubseteq C \oplus B \\ A\&C \sqsubseteq B\&C & C\&A \sqsubseteq C\&B \end{array}$$

$$\begin{array}{ll} A \oplus A = A & A\&A = A \\ A \oplus B = B \oplus A & A\&B = B\&A \\ A \oplus (B \oplus C) = (A \oplus B) \oplus C & A\&(B\&C) = (A\&B)\&C \end{array}$$

$$\begin{array}{rcl} A\&(B \oplus C) & = & (A\&B) \oplus (A\&C) \\ A \oplus (B\&C) & = & (A \oplus B)\&(A \oplus C) \end{array}$$

Moreover

$$\begin{array}{ll} A\|B & = B\|A \\ A\|(B\|C) & = (A\|B)\|C \\ < s > \|A & = A \\ \hline A\|(B \oplus C) & = A\|B \oplus A\|C \\ A\|(B\&C) & = A\|B\&A\|C \end{array}$$

If we consider the set $\mathcal{P}$ of the processes (considered as sentences), and according to the definition of a lattice-ordered monoid (shortly, lo-monoid), then $(\mathcal{P}; \|, \oplus, \&)$ is such a lo-monoid.

An executional deductive framework is described in Ito's paper, and it is explained how to "execute" a logical sentence. Unfortunately, it is not easy to find the computational (operational) aspect of this logic.

In this paper we use dynamic logic to reason over the executions of nondeterministic choices. This logic is a modal logic whose modality corresponds to the execution of actions. We use the formula $< do_p(\alpha) > \varphi$ which could be used to express the capability or feasibility of a process, as well as the effective action and correctness. $< do_p(\alpha) > \varphi$ has the meaning that all prerequired conditions of action α are satisfied, and as a result of executing α by process p, φ holds.

In fact we try to express how a machine (a nondeterministic language compiler, for instance) is "reasoning" whether or not ϕ holds as a result of performing one of the nondeterministic choices. Regarding a choice $(a\&b)$ made by the machine, it has an angelic behaviour. Regarding a choice $(a \oplus b)$ made by the computational environment, the machine has no influence on which action is chosen; it must be prepared to deal with all of the possible actions or events - in this way the computational environment shows a demonic behaviour.

2 Angelic and demonic nondeterminism

We simulate sometimes how the machine is reasoning (try to imagine that you wish to implement a nondeterministic language, for instance) whether φ holds as a result of performing on internal or external nondeterministic choice. When we use the word "machine" we refer to an execution of a program (a compiler, for instance). A programmer using a (nondeterministic) programming language thinks in a different way than a programmer who have written the compiler for that programming language. In the first case, a programmer thinks about an internal nondeterminism in a deterministic way, even when the nondeterministic constructions of the language are used; usually a programmer simulates in a deterministic way the computation associated to a nondeterministic program. Moreover, the programmer expects that the used compiler has an angelic behaviour, namely if there is at least one successful computational path defined by his program, then the execution of the compiled program will succeed. This expectation is transmitted to the compiler which is looking to its

internal nondeterminism as performing in an angelic way. Therefore, regarding to an internal choice $\alpha\&\beta$, the machine thinks of itself as showing an angelic behaviour: i.e. whenever it is possible to perform an action in the correct (desired) way, it will choose to perform that action (in the correct way). Regarding to an external choice $\alpha \oplus \beta$, the machine(compiler) has no influence on which action is chosen, and it must be prepared to deal with either of the actions; in particular machine may not assume that the external environment will make the best choice - i.e. environment shows a demonic behaviour. The author is thinking that the ambiguity of the computational context associated to the notion of nondeterminism is the main reason why the latter notion is unclear. As you remarked, we already discussed about determinism, angelic and demonic nondeterminisms considering various points of view related to a nondeterministic programming language, and considering a certain computational space. An exhaustive study of nondeterminism is still waiting to be done; this article may be a step to such a comprehensive approach.
Our computational space was defined, and we reason as a compiler (or an implementer simulating the actions of the desired compiler). According to the previous remarks, our formal approach is given by the following two formulas:

$$< do_p(\alpha\&\beta) > \varphi \equiv < do_p(\alpha) > \varphi \vee < dp_p(\beta) > \varphi$$
$$< do_p(\alpha \oplus \beta) > \varphi \equiv < do_p(\alpha) > \varphi \wedge < do_p(\beta) > \varphi$$

There is an interaction between a machine and its computational environment, and we reflect this by defining a synchronization between the internal and external forms of nondeterminism. As usual, synchronization is a operation of composing two systems by identifying, relating some of their actions. In our case the result of such a synchronization is an execution; our aim is to have some judgements about this execution. We consider an operator which is similar to the "concurrent execution" of the Ito's logic of execution; the set Act of the possible actions of the machine and the environment together with this operator $\|$ becomes a monoid $(Act, \|, u)$. Moreover this operator isdefined also by $\alpha\|\beta = (\alpha \oplus \beta)\|(\alpha\&\beta)$ (by definition), and

$$< do_p(\alpha \cdot \beta) > \varphi =< do_p(\alpha) > \varphi \vee < do_p(\beta) > \varphi.$$

We show that $(Act, \&, \oplus, u)$ is a lattice with the least element u,
and $(Act, \|, \&, \oplus, u)$ is a lattice-ordered monoid where $\alpha\|\beta = (\alpha \oplus \beta)\|(\alpha\&\beta)$ holds also. In order to study this algebraic structure we use the notation $(Act, \|, \wedge, \vee, u)$ instead of $(Act, \|, \&, \oplus, u)$.

3 The lattice-ordered monoid of actions

We consider the algebraic system $(Act, \|, \wedge, \vee, u)$ where $(Act, \|, u)$ is a monoid, $(Act, \wedge, \vee, u)$ is a lattice with the least element u, and for all $a, b, c \in Act$ we have
$a\|(b \vee c) = (a\|b) \vee (a\|c)$,

$a\|(b \wedge c) = (a\|b) \wedge (a\|c)$,
$a\|b = (a \vee b)\|(a \wedge b)$.

We assume also that if the monoid $(Act, \|, u)$ has a zero z, then z is the greatest element in the lattice $(Act, \wedge, \vee, u)$.
It is easy to see that the monoid $(Act, \|, u)$ is commutative. We have also the following elementary properties:

Proposition 1 *If $a, b, c \in Act$ then*

i) $a\|b \geq a \vee b$;

ii) if $a \leq b$, *then* $a\|c \leq b\|c$;

iii) if $a, b \leq c$ *and* $a \wedge b = u$, *then* $a\|b \leq c$;

iv) if $a\|b = a \vee b$ *and* $c \leq b$, *then* $a\|b = b \vee (a\|c)$.

In what follows the notation $a_1 ... \overline{a_i} ... a_m$ will mean $a_1\|a_2 ... \|a_{i-1}\|a_{i+1}\| ... a_n$.

Proposition 2 *Let be* $a_i \in Act, 1 \leq i \leq n$. *Then*

i) $\|_{i=1}^n a_i = (\vee_{i=1}^n a_1 ... \overline{a_i} ... a_n)\|(\wedge_{i=1}^n a_i) = (\wedge_{i=1}^n a_1 ... \overline{a_i} ... a_n)\|(\vee_{i=1}^n a_i)$.

ii) If $a_i\|a_j = a_i \vee a_j$ *for every* $1 \leq i, j \leq n$, *then* $\|_{i=1}^n a_i = \vee_{i=1}^n a_i$.

Proof :

i) We will prove the first equality by induction. For n=2 the result holds by previous results. Suppose it is true for an arbitrary fixed $n \in N$. We have $\vee_{i=1}^{n+1} a_1 ... \overline{a_i} ... a_{n+1}) = a_{n+1}\|(\vee_{i=1}^n a_1 ... \overline{a_i} ... a_n) \vee \|_{i=1}^n a_i = a_{n+1}\|(\vee_{i=1}^n a_1 ... \overline{a_i} ... a_n) \vee (\vee_{i=1}^n a_1 ... \overline{a_i} ... a_n)\|(\wedge_{i=1}^n a_i) = (\vee_{i=1}^n a_1 ... \overline{a_i} ... a_n)\|(a_{n+1} \vee \wedge_{i=1}^n a_i)$ and then $(\vee_{i=1}^{n+1} a_1 ... \overline{a_i} ... a_{n+1})\|(\wedge_{i=1}^{n+1} a_i) = (\vee_{i=1}^n a_1 ... \overline{a_i} ... a_n)\|(a_{n+1} \vee \wedge_{i=1}^n a_i)\|(a_{n+1} \wedge \wedge_{i=1}^n a_i) = (\vee_{i=1}^n a_1 ... \overline{a_i} ... a_n)\|(a_{n+1}\| \wedge_{i=1}^n a_i) = (\vee_{i=1}^n a_1 ... \overline{a_i} ... a_n)\|(\wedge_{i=1}^n a_i)\|a_{n+1} = (\|_{i=1}^n a_i)\|a_{n+1} = \|_{i=1}^{n+1} a_i$. The second equality can also be proved by induction.

ii) By induction.

Proposition 3 *For every* $a, b_i \in Act, 1 \leq i \leq n$, *we have* $a \wedge \|_{i=1}^n b_i = a \wedge \|_{i=1}^n (a \wedge b_i)$.

Proof : $a \wedge \|_{i=1}^n (a \wedge b_i) = a \wedge [a\| ... \|a \wedge a\|(\wedge_{i=1}^n b_i) \wedge ... \wedge a\|(\wedge_{i=1}^n b_1 ... \overline{b_i} ... b_n) \wedge \|_{i=1}^n b]$ $= a \wedge \|_{i=1}^n b_i$.

Corollary 1 *If* $a, b_i \in Act$ *and if* $a \wedge b_i = u$ *for every* i, $1 \leq i \leq n$, *then* $a \wedge \|_{i=1}^n b_i = u$.

We have also some structural properties which connect the subjacent structures of our lattice-ordered monoid.

Proposition 4 *Let be $a \in Act$. The set $s(a) = \{b \in Act \mid a\|b = a\}$ is a lattice ideal and a submonoid of Act; if $a\|a = a$ then $s(a) = (a]$.*

Proof : A lattice ideal can be characterized by: if $b, c \in s(a)$, than $b \bigvee c \in s(a)$, and if $b \in s(a)$ and $c \leq b$ then $c \in s(a)$. It is easy to show that $s(a)$ is a semilattice. Now let be $b \in s(a)$, $c \in Act$ and $c \leq b$. If $a\|b = a$ then $b \leq a$, and thus $a = a \bigvee b = a\|b$. By previous results $a = a\|b = b \bigvee (a\|c)$ which shows that $a\|c \leq a$. But $a\|c \geq a$, which forces $a\|c = a$, hence $c \in s(a)$. Straightforward calculations show that $s(a)$ is a submonoid of Act. Clearly, $s(a) \subseteq (a]$. Let be $x \in (a]$; then $a\|x = (a \bigvee x)\|(a \bigwedge x) = a\|(a \bigwedge x) = (a\|a) \bigwedge (a\|x) = a \bigwedge (a\|x) = a$ and thus $x \in s(a)$.

Proposition 5 *Let be $a \in Act$. The set $p(a) = \{b \in Act \mid a \bigwedge b = u\}$ is a lattice ideal and a submonoid of Act.*

Proof : By the previous corollary, $p(a)$ is a submonoid of Act. Let be $b, c \in p(a)$; then $a \bigwedge (b \bigvee c) \leq (a \bigwedge (b\|c) = u$ by the same corollary again, showing that $p(a)$ is a semilattice. On the other hand, if $b \in p(a), c \in Act$ and $c \leq b$ then $u \leq a \bigwedge c \leq a \bigwedge b = u$, hence $c \in p(a)$.

Proposition 6 *i) Every filter of the lattice $(Act, \bigwedge, \bigvee, u)$ is an ideal of the monoid $(Act, \|, u)$.*

ii) If $(Act, \|)$ is a principal ideal semigroup, then $[a) = a\|Act, \forall a \in Act$.

iii) If $I \subset Act$ is a prime semigroup ideal, then $Act - I$ is a sublattice of $(Act, \bigwedge, \bigvee, u)$.

Proposition 7 *If $(Act, \|, u)$ is with cancellation, then $(Act, \bigwedge, \bigvee, u)$ is a distributive lattice.*

The lack of cancellation leads to the study of the idempotents of the monoid. The study of this lo-monoid leads to an interesting decomposition theorem; this result is interesting on its own, and it is the subject of another paper. We selected only those results which are appropriate for our approach concerning nondeterminism.

4 The nondeterministic degree

We dare to remember that if $(L, \bigwedge, \bigvee)$ is a lattice, then v: $(L, \bigwedge, \bigvee) \to (R, +)$ is a valuation if $v(a \vee b) + v(a \wedge b) = v(a) + v(b)$. Moreover, if v is strictly isotone (i.e. if $x < y$ then $v(x) < v(y)$)), then $(L, \bigwedge, \bigvee)$ is called a metric lattice. A well-known theorem tells that every metric lattice is modular.

Proposition 8 *Let* $(Act, \|, \&, \oplus, u)$ *be our lo-monoid. If we consider a* $ng : (Act, \|) \to ([0,1], +, 0)$ *- where* $([0,1], +, 0)$ *is a monoid - such that* ng *is a strict isotone monoid homomorphism, then* $(Act, \&, \oplus, u)$ *is a modular lattice.*

Proof : We define the nondeterministic degree ng by $ng \equiv < do_p - > \varphi : Act \to ([0,1], +)$. In this way we relate the nondeterministic degree to the process (or processor) p, and to the property φ.
We have

$ng(\alpha \| \beta) = ng(\alpha) + ng(\beta)$, and

$\alpha \| \beta = (\alpha \oplus \beta) \| (\alpha \& \beta)$,

then we have also $ng(\alpha \| \beta) = ng((\alpha \oplus \beta) \| (\alpha \& \beta)) = ng(\alpha \oplus \beta) + ng(\alpha \& \beta)$. This means that $ng(\alpha) + ng(\beta) = ng(\alpha \oplus \beta) + ng(\alpha \& \beta)$, i.e. ng is a valuation. ng is also strict isotone, and it follows that $(Act, \&, \oplus, u)$ is a metric lattice. Since any metric lattice is modular, $(Act, \&, \oplus, u)$ is a modular lattice.

Other results:

1. From $ng(\alpha \| \beta) = ng(\alpha) + ng(\beta)$, considering $\beta = u$, then $ng(\alpha) = ng(\alpha) + ng(u)$, i.e. $ng(u) = 0$

2. We don't have a zero in $(Act, \|, u)$. If we suppose that there is a zero z, then $ng(\alpha \| z) = ng(\alpha) + ng(z)$, i.e. $ng(z) = ng(\alpha) + ng(z)$, i.e. $ng(z) = \infty \notin [0,1]$

3. If $\alpha \| \beta = \alpha \oplus \beta$ i.e. we have only one form of nondeterminism which is active in a synchronization, then $ng(\alpha \| \beta) = ng(\alpha) + ng(\beta) = ng(\alpha \oplus \beta) + ng(\alpha \& \beta) = ng(\alpha \oplus \beta)$, i.e. $ng(\alpha \& \beta) = 0$. We have also that $u \leq \alpha \& \beta$, and this means that $0 = ng(u) \leq ng(\alpha \& \beta) = 0$.
Since ng is strictly isotone, $ng(\alpha \& \beta) = 0$ iff $\alpha \& \beta = u$.
Therefore if there is such a nondeterministic degree $ng : (Act, \|, u) \to ([0,1], +, 0)$ which is a strict isotone monoid homomorphism, then $\alpha \| \beta = \alpha \oplus \beta$ if and only if $\alpha \& \beta = u$.

Proposition 9 *If* $ng : (Act, \|, u) \to ([0,1], +)$ *is also injective, then* $(Act, \&, \oplus)$ *is a distributive lattice.*

Proof : We consider $\alpha, \beta, \delta \in Act$ such that (*): $\alpha \& \beta = \alpha \& \delta$, and (**): $\alpha \oplus \beta = \alpha \oplus \delta$.
$(Act, \&, \oplus)$ is a distributive lattice if and only if ((*) and (**) imply $\beta = \delta$).
From (**) we have $ng(\alpha \oplus \beta) = ng(\alpha \oplus \delta)$, i.e. $ng(\alpha) + ng(\beta) - ng(\alpha \& \beta) = ng(\alpha) + ng(\delta) - ng(\alpha \& \delta)$. According to (*) we have $ng(\alpha \& \beta) = ng(\alpha \& \delta)$. Therefore $ng(\beta) = ng(\delta)$, and $\beta = \delta$ - because ng is injective.

References

[1] T. Ito: *Logic of Execution*, TACS '91, LNCS 526, Springer-Verlag, 1991.