ELSEVIER Theoretical Computer Science 139 (1995) 243–273

Theoretical Computer Science

Computable concurrent processes☆

Yiannis N. Moschovakis

Department of Mathematics, University of California, Los Angeles, CA 90024, USA

Received July 1992; revised April 1994
Communicated by G. Jäger

Abstract

We study relative computability for processes and process transformations, in general, and in particular the non-deterministic and concurrent processes which can be specified in terms of various *fair merge* constructs. The main result is a normal form theorem for these (relatively) *computable process functions* which implies that although they can be very complex when viewed as classical set-functions, they are all "loosely implementable" in the sense of Park (1980). The precise results are about the *player model* of concurrency introduced in Moschovakis (1991), which supports both fairness constructs and full recursion.

Consider the nondeterministic process[1] C defined recursively by the equation

$$C = (0 \text{ or } 1); C, \tag{1}$$

where 0 and 1 stand for the acts of printing the corresponding digits, ";" denotes *sequential execution* and "*or*" stands for unrestricted (random, autonomous) *choice*. Intuitively, an execution of C prints a 0 or a 1, at random, and then calls itself to repeat this, so "in the end" the *trace* of C produced by this sequence of random choices is (essentially) some infinite, binary sequence. We would like to call C a *computable process*, since it is defined by such an elementary recursion from the simple *or* construct, but this brings up a question: the traces of C are arbitrary binary sequences, most of them not individually computable.

☆ A preliminary version of these results was announced in the 1990 POPL meeting [12]. The POPL announcement contained some additional material on the semantics of concurrent languages, whose proofs have been written up in [13]. During the preparation of this paper the author was partially supported by an NSF Grant.

[1] Terms like "process," "execution," "trace," "fairness", etc. are used intuitively and vaguely in these introductory remarks. One of the main aims of "concurrency theory" is to supply rigorous definitions of these notions within a specific model, and we will review how this is done for our model in Section 1.

SSDI 0304-3975(94)00117-2

The same question arises more dramatically when we combine non-determinacy with interaction, as in the following classical example of Park [15]: set

$$P = X := 0;\ Y := 0;\ \mathit{while}(X = 0)\{X := 1 \parallel Y := Y + 1\}, \tag{2}$$

where the parallel construct $\parallel$ is meant to be understood *fairly*, so that both of its process arguments will get as many chances to execute as they need. In fact, the simple assignment $X := 1$ needs only one chance, which must be given to it "at some point", after which there is nothing more to be done and the process terminates. Thus – Park argues – the traces of P are all finite sequences of acts of the form

$$\langle X := 0,\ Y := 0,\ \underbrace{Y := Y + 1, \ldots, Y := Y + 1}_{n-1},\ X := 1 \rangle,$$

and any satisfactory semantics for such *concurrent processes* should entail this basic fact about P. Using Park's fair-merge construct, we can also set

$$M = (\bar{0} \parallel \bar{1}), \tag{3}$$

where $\bar{0} = \langle 0, 0, \ldots, \rangle$ and $\bar{1} = \langle 1, 1, \ldots, \rangle$ are the constant infinite sequences. The traces of M are exactly the "fair" binary sequences, those with infinitely many 0's and infinitely many 1's, most of them, again, not individually computable. Still, we are tempted to say that M is a computable process, relative to the fair merge construct.

Park [15] was responding to Dijkstra [5], who advocated banning definitions like (2) and (3) from programming because they involve *unbounded non-determinism* – which "cannot be implemented" – and he argued that in discussing implementability issues we should understand nondeterministic and concurrent definitions of this type *loosely*:

> No-one requires of a correct implementation for parallelism that there be an appropriate sense in which *all* scheduling algorithms be possible in it, only that there be one such scheduling algorithm, and if fairness be required that the scheduler be fair.

So we can claim that C and M are implementable since the computable, alternating sequence of 0's and 1's is a trace of both of them. Now the existence of implementations is obviously important, but it will hardly do as a sole criterion for computability: no-one would call an arbitrary set of infinite binary sequences computable simply because it may happen to have a computable member. If there is a natural concept of *computable process*, then it must involve more than the mere existence of a computable trace.

At the same time, we cannot profitably interpret the definitions of C or M so that they only have computable traces, we truly need *all* their traces, if we are to gain anything from allowing unrestricted choice or fair merging among our primitive programming constructs. When we call M from within some program E, we are simply specifying that a fair, binary sequence should be produced; and in proving

properties of E, all we can assume is that the sequence produced will be fair, not that it is also computable, as it may well be that some piece of hardware is doing the merging, noncomputably. In fact, we tend to call nondeterministic processes like C, P and M "computable" primarily because they are defined in terms of simple and intuitive programming constructs. The fact that they have computable traces should be a formal consequence of such definitions (however trivial for these examples) and not the basic reason for thinking them computable.

The main aim of this paper is to define rigorously and establish some basic facts about *computable processes* and *computable process functions*, in the context of the game-theoretic modeling of concurrent, asynchronous systems introduced in [13]. We will characterize these objects in the cases where we take as given unrestricted choice, the full (unfair) merge, the fair merge in the sense of Park (as above) and the richer *state-dependent fair merge* which is most natural in the context of our concurrency theory. The main result is that the processes which are computable relative to these constructs are all "loosely implementable" in the sense of Park, although their sets of traces can be extremely complex; it is a corollary of a Normal Form Theorem for computable processes and process functions, which also implies that these notions are very robust and provides some considerable justification for our choice of definitions.

There are two important notions in *the player model* of concurrency introduced in [13], the *players* which model *concurrent processes* and the *implemented player functions* (*ipfs*) which represent *process transformations*. After a brief review of these notions in Section 1 and a briefer comparison of the player model with other modelings of concurrency, we will define process computability and state rigorously our results in Section 2. These sections can be read with only a fair acquaintance of Sections 1–4 and 6 of [13], but the proofs, in the remaining two sections, depend heavily on the technical machinery developed in [13].

1. The player and other models of concurrency

A useful tool for defining the player model and comparing it with other modelings of concurrency is the following, simple formal language.[2]

1.1. The expressions (terms) of $\mathrm{FLR}_0(\tau)$ are defined inductively by

$$E ::= x \mid \mathtt{f}(E_1, \ldots, E_n) \mid \mathtt{rec}(x_1, \ldots, x_n)[E_0, \ldots, E_n],$$

where x is any variable (from a fixed, infinite set), $\mathtt{f}$ is any function symbol (from a fixed vocabulary τ of function symbols, each with an assigned, nonnegative arity) and

[2] This was called $\mathscr{L}$ in [13]. FLR_0 is the equational (or "propositional") part of FLR, the *formal language of recursion* introduced in [11]. There is also an intermediate language FLR_1, which comes from FLR by dropping the conditional but retaining value passing and functional recursion, and which plays for recursion the role that the *first-order predicate calculus* plays for explicit definability.

a more familiar notation for the *mutual recursion construct* is

$$E_0 \text{ where } \{x_1 = E_1, \ldots, x_n = E_n\}.$$

FLR_0 is an abstract version of several languages which have been used to study concurrency, including Hoare's *communicating sequential process* (CSP) [7], Milner's *calculus of communicating systems* (CCS) [9], the metric approach of de Bakker and Zucker [4] and the *reactive processes* of Manna and Pnueli [8]. The idea is that a particular model can be made precise and compared to other models by giving rigorous semantics for $\mathrm{FLR}_0(\tau)$, with some specific vocabulary τ. Most of these languages include some or all of the following function symbols, in common, infix notation and with their intended, intuitive meaning:

skip	skip, or idle,
a	act execution,
ax	act prefixing,
$x;y$	sequential execution,
if R then x else y	conditional execution,
x or y	autonomous choice,
$x \parallel y$	merge.

The act symbols are from some fixed set A (sometimes endowed with additional structure) and the conditionals are often described by an auxiliary language of "conditions" (on some assumed "state"). When included, the merge operation is sometimes interpreted by the full (unfair) rather than the fair merge. In addition, most theories include special functions to model the *interaction* (or *synchronization*) of processes, which is the heart of the problem of modeling concurrency. There is no single, special construct for interaction in the player model, the ability to interact is, in some sense, built into the very definition of "process", and it is controlled by a typing mechanism for interaction.

In [13] we defined three kinds of $\mathrm{FLR}_0(\tau)$ semantics, for structures of the form

$$\mathfrak{A} = (\mathscr{S}, \mathscr{I}) = (\mathsf{States}, \iota, \mathsf{Acts}, \mathit{skip}, \mathit{exec}, \mathscr{I}), \tag{4}$$

where $\mathscr{S} = (\mathsf{States}, \iota, \mathsf{Acts}, \mathit{skip}, \mathit{exec})$ is the *state structure* of $\mathfrak{A}$ and $\mathscr{I}$ is a set of "functions" which distinguishes the semantics. About $\mathscr{S}$ we assume the following: States is an arbitrary set of *states* containing the initial state ι; Acts is an arbitrary set of *atomic acts* containing the "delay" act *skip*; every act a induces a function $s \mapsto sa = \mathit{exec}(s, a)$ on the states, such that $s\ \mathit{skip} = s$; and every state s is *accessible* by a sequence of acts $a_1, \ldots, a_n$ from ι, i.e. $s = a_1 a_2 \cdots a_n \iota$. A state structure $\mathscr{S}$ is *trivial* if its only member is ι, in which case we can identify it with the set of its acts. We summarize briefly and comment on each of these interpretations.

1.2. Input-dependent, deterministic processes

A *procedure* (or *input-dependent stream*, [13, 2.4–2.8]) on $\mathscr{S}$ is a function α: States→Streams(Acts) which assigns to each state a stream of acts. These are the natural denotations of programs which read the state just once (presumably to get input) and then execute a stream of acts in total isolation from the environment. The set $\Pi=\Pi(\mathscr{S})$ of procedures over $\mathscr{S}$ is a directed, complete poset (dcpo) and in a *procedure structure* $(\mathscr{S},\mathscr{I})$ for $\mathrm{FLR}_0(\tau)$, $\mathscr{I}$ assigns to each n-ary function symbol f of the vocabulary τ a continuous, n-ary function $\mathscr{I}(\mathbf{f}):\Pi^n\to\Pi$. To interpret $\mathrm{FLR}_0(\tau)$ on $(\mathscr{S},\mathscr{I})$, we let the variables range over procedures and we use composition and least-fixed-point recursion to associate with each expression E and each sequence x of n variables (which includes all the free variables of E) an n-ary procedure function

$$\text{procedure}(\mathfrak{A},x)E=\phi_E:\Pi^n\to\Pi.$$

Two expressions E and M are *procedure equivalent* if for all procedure structures, $\phi_E=\phi_M$. There are obvious, procedure interpretations of act execution, act prefixing and sequential and conditional execution [13, 2.7].

In a trivial state structure, procedures stand for totally isolated programs which cannot interact at all with one another. If the state is nontrivial, then the sequential execution $\alpha;\beta$ affords some minimal communication at the entry and exit stages, because if $\alpha(s)$ terminates after executing the acts $a_1,\ldots,a_n$, then (after these acts) $(\alpha;\beta)(s)$ executes the stream $\beta(a_1a_2\cdots a_ns)$ which may depend on $a_1,\ldots,a_n$.

1.3. Interactive, deterministic processes

A *behavior* (or *state-dependent stream of acts*) is a partial strategy for II in the *interaction game* pictured in Fig. 1, formally a partial function

$$\sigma:\mathsf{States}^*\to\mathsf{Acts}\times\{\partial,\mathbf{t}\} \tag{5}$$

on nonempty sequences of states, except that we identify behaviors which agree on all partial runs of the game [13, Section 3]. Player II represents in this game the denotation of some deterministic, interactive program, which responds (if defined) to each states s_n played by I with an act a_n and either the indicator ∂, meaning that more moves are needed, or **t**, signifying successful termination; player I represents "the rest of the world", the collective action of all the other agents operating in the same

I	s_0	s_1	s_2	$\cdots$	s_n
II	(a_0,∂)	(a_0,∂)	(a_2,∂)	$\cdots$	$(a_n,\mathbf{t})$
State: ι	s_0a_0	s_1a_1	s_2a_2	$\cdots$	n_3a_3

Fig. 1. The interaction game; a terminating run.

environment. The set $\mathsf{B}=\mathsf{B}(\mathscr{S})$ of behaviors is a dcpo. An n-ary *behavior function* is any continuous $F:\mathsf{B}^n\to\mathsf{B}$, and a *behavior structure* $\mathfrak{A}=(\mathscr{S},\mathscr{I})$ is one where $\mathscr{I}$ assigns behavior functions (of the appropriate arity) to the function symbols. The *behavior semantics* of such a structure interpret recursion by the taking of least fixed points, and they associate with each expression E and each list of n variables $\boldsymbol{x}$ which includes all the free variables of E, a behavior function

$$\text{behavior}(\mathfrak{A},\boldsymbol{x})E=F_E:\mathsf{B}^n\to\mathsf{B},$$

the *behavior denotation* of E. Two expressions E and M are *behavior equivalent* if $F_E=F_M$ on every behavior structure.

Behaviors are the natural denotations of deterministic, interactive programs and several examples of them are worked out in [13, Section 3], including act execution and prefixing, sequential and conditional execution and the handling of interrupts. It is also claimed in [13, Theorem 3.7] that *procedure equivalence coincides with behavior equivalence* for FLR_0 expressions; the proof of this is quite simple.

1.4. Interaction through the state vs. message passing

Among actual, interactive systems, there is an important difference between those with a common, central memory which can be accessed by all the processes, and those in which communication can only take place by the passing of messages through private channels. Our notion of behavior might suggest that the player model can only handle efficiently central-memory interaction, but this is not true. Suppose, for example, that the process x can only execute acts in some set $K\subsetneq\mathsf{Acts}$, the process y can only "see" changes in the state which are produced by the execution of acts in K, and no other process can either execute acts in K or discern changes in the state produced by the execution of acts in K; we might then reasonably call K a *private, one-way channel* of communication from x to y. There is a natural notion of *behavior type* for processes which provides a flexible mechanism for introducing and keeping track of such restrictions and by which we can easily represent in the player model a *local state* interaction through message-passing [13, 4.3–4.7, 8.2]. See also [13, 3.6] for some remarks on the representation of concurrent systems by the game of interaction.

1.5. Traces of behaviors

If ω is a (total) *strategy for player* I and $\sigma\in\mathsf{B}$, then

$$\omega*\sigma=a_0,a_1,\ldots \tag{6}$$

is the stream of acts "executed" (by player II) in the run of the game where I plays by ω and II responds by σ – terminated with $\mathbf{t}$ if II ever plays $(a_n,\mathbf{t})$. The *trace* of σ is the set of all these streams,

$$\mathit{trace}(\sigma)=\{\omega*\sigma\mid\omega\text{ is a strategy for I}\}. \tag{7}$$

If, for example,

$$\begin{aligned}&\sigma(\langle s_0\rangle)=(a,\partial),\\&\sigma(\langle s_0,s_1\rangle)=\textit{if } R(s_0) \textit{ then } (b,\mathbf{t}) \textit{ else } (c,\mathbf{t}),\end{aligned}\tag{8}$$

then $trace(\sigma)=\{\langle a,b,\mathbf{t}\rangle,\langle a,c,\mathbf{t}\rangle\}$ is a doubleton and an "observer" who is simply recording the acts executed by σ (in repeated experiments) without knowing the state structure or the relation R may well consider σ a nondeterministic process.

1.6. Concurrent processes

A *player* on a state structure $\mathscr{S}$ is any nonempty set of behaviors, and we set

$$\mathscr{P}(\mathscr{S})=\mathscr{P}=\{x\mid \emptyset\neq x\subseteq \mathsf{B}(\mathscr{S})\};\tag{9}$$

this is the set by which we model the nondeterministic, interactive processes on $\mathscr{S}$. Intuitively, a player x can "play" (interact with the world, be implemented by) any of its "behaviors", i.e. any $\sigma\in x$. A player x *is deterministic* if it is a singleton $x=\{\sigma\}$ and *total* if every $\sigma\in x$ is a totally defined strategy. For example, for each act a.

$$a^{J}=\{\lambda(\langle s_0,\ldots,s_n\rangle)(a,\mathbf{t})\}\tag{10}$$

is the total, deterministic player who (confronted by any state) executes a and quits. The *trace* of a player is the union of the traces of its behaviors,

$$\begin{aligned}trace(x)&=\bigcup\{trace(\sigma)\mid\sigma\in x\}\\&=\{\omega*\sigma\mid\sigma\in x,\omega \text{ is a strategy for I}\}.\end{aligned}\tag{11}$$

Now $\mathscr{P}$ is not a dcpo in any natural way, so it is not immediate how to model process transformations by functions which have fixed points. Before dealing with this, notice that each behavior function $F:\mathsf{B}^n\to\mathsf{B}$ induces naturally a (set) function $F^{J}:\mathscr{P}^n\to\mathscr{P}$ by "distribution",

$$F^{J}(x_1,\ldots,x_n)=\{F(\sigma_1,\ldots,\sigma_n)\mid\sigma_1\in x_1,\ldots,\sigma_n\in x_n\};\tag{12}$$

for example (and skipping the J),

$$\begin{aligned}&x;y=\{\sigma;\tau\mid\sigma\in x,\tau\in y\},\\&\textit{if } R \textit{ then } x \textit{ else } y=\{\textit{if } R \textit{ then } \sigma \textit{ else } \tau\mid\sigma\in x,\tau\in y\}\\&\qquad\qquad=\{F_R(\sigma,\tau)\mid\sigma\in x,\tau\in y\},\end{aligned}$$

where for each σ and τ,

$$F_R(\sigma,\tau)=\lambda(\langle s_0,\ldots,s_n\rangle)\ \textit{if } R(s_0) \textit{ then } \sigma(\langle s_0,\ldots,s_n\rangle) \textit{ else } \tau(\langle s_0,\ldots,s_n\rangle).$$

These liftups are *deterministic player functions*, in the sense that when applied to deterministic arguments (singletons) they yield deterministic values. Typical

nondeterministic functions on $\mathscr{P}$ include

$$x \text{ or } y = x \cup y, \tag{13}$$

$$x + y = \{\text{if } R \text{ then } \sigma \text{ else } \tau \mid \sigma \in x, \tau \in y, R \subseteq \mathsf{States}\}, \tag{14}$$

which represent two distinct ways of modeling *choice*. Notice here that for all players x, y, easily,

$$\mathit{trace}(x+y) = \mathit{trace}(x \text{ or } y) = \mathit{trace}(s) \cup \mathit{trace}(y),$$

but in general, $(x \text{ or } y) \neq (x+y)$, in fact

$$a^J \text{ or } b^J \neq a^J + b^J$$

if a and b are distinct acts and the state structure is nontrivial: because if $R(s)$ is true for some state and false for some other, then the state-dependent strategy

$$\sigma(s_0) = \text{if } R(s_0) \text{ then } (a, \mathbf{t}) \text{ else } (b, \mathbf{t})$$

is a behavior of $a^J + b^J$ but not of a^J or b^J. For the same reason,

$$a^J; (b^J + c^J) \neq a^J; b^J + a^J; c^J \tag{15}$$

for distinct acts in a nontrivial state structure, although these two players have the same traces. Thus, *a non-deterministic player is not determined by its trace.*

1.7. Modeling process transformations

One of the basic premises of [13] is that a function $f: \mathscr{P} \to \mathscr{P}$ models a *process transformation* if it is determined by its "implementations", and one is tempted (at first) to consider only the *linear implementations* of f, i.e. the continuous $F: \mathsf{B} \to \mathsf{B}$ such that

$$\sigma \in x \;\Rightarrow\; F(\sigma) \in f(x).$$

But if the recursive definition (with a parameter)

$$y = x; y \tag{16}$$

makes sense (as it should), we would expect it to define the process transformation

$$y(x) = \{\sigma_0; \sigma_1; \cdots \mid \sigma_0, \sigma_1, \ldots \in x\}$$

and the most natural implementation of $y(x)$ is the "infinitary" continuous operation

$$Y(\sigma_0, \sigma_1, \sigma_2, \ldots) = \sigma_0; \sigma_1; \sigma_2 \cdots.$$

In general, a (infinitary) behavior function (of n arguments) is any continuous operation

$$F: (N \to \mathsf{B})^n \to \mathsf{B} \quad (N = \{0, 1, \ldots\}),$$

and we call F an *abstract implementation* of a function

$$f: \mathscr{P}^n \to \mathscr{P}$$

if for every n-tuple of players $\boldsymbol{x}=x_1,\ldots,x_n$ and sequences of behaviors $\boldsymbol{p}=p_1,\ldots,p_n$,

$$p_1:N\to x_1,\ldots,p_n:N\to x_n \Rightarrow F(\boldsymbol{p})\in f(\boldsymbol{x}).$$

A function $f:\mathscr{P}^n\to\mathscr{P}$ is *implementable* if there exists some set I of abstract implementations which determines its values, i.e.

$$f(\boldsymbol{x})=\{F(\boldsymbol{p})\mid p_1:N\to x_1,\ldots,p_n:N\to x_n, F\in I\}. \tag{17}$$

The second basic premise of [13] is that the correct way to model process transformations is to forget the extensional "implementable" and use the intensional "implemented" [13, Section 7]: an n-ary *implemented player function* (*ipf*) is[3] any set f of n-ary abstract implementations which is closed under a suitable reducibility relation [13, 7.2–7.4]; and the (player) value of f is given by

$$f(\boldsymbol{x})=\{F(\boldsymbol{p})\mid p_1:N\to x_1,\ldots,p_n:N\to x_n, F\in f\}.$$

A set $I\subseteq f$ *generates* an ipf f, if

$$f=[I]=\{F\mid F \text{ is reducible to some } G\in I\}, \tag{18}$$

i.e. if f is the closure of I under reducibility. It can be verified that every set I which generates f determines the values of f by (17) [13, 7.3]. Most often, we define ipfs by specifying a simple generating set for them. For example, for each behavior function $F:\mathbf{B}^n\to\mathbf{B}$, we let

$$F^{\prime}=[F^*] \text{ with } F^*(p_1,\ldots,p_n)=F(p_1(0),\ldots,p_n(0)), \tag{19}$$

the notation chosen because this $F^{\prime}$ indeed satisfies (12). Similarly,

$$or=[F_l,F_r] \quad \text{with } F_l(p,q)=p(0), \qquad F_r(p,q)=q(0), \tag{20}$$

is the ipf version of autonomous choice, easily satisfying

$$x \text{ or } y=\{F_l(p,q)\mid p:N\to x,q:N\to y\}\cup\{F_r(p,q)\mid p:N\to x,q:N\to y\}.$$

A similar modeling can be given for $x+y$ as an ipf, but now we need an infinite generating set, all the conditionals. An ipf is *deterministic* if it is generated by a single behavior function F_0, as each $F^{\prime}$ is but *or* is not.

1.8. Merge operations

A *merger* on a process structure $\mathfrak{A}$ is any (total) function $\mu:\mathsf{States}^*\to\{0,1\}$ on nonempty sequences of states to $\{0,1\}$. Given behaviors σ_0,σ_1 in $\mathfrak{A}$, $\mu[\sigma_0,\sigma_1]$ is the (conjunctive) *merged behavior* of σ_0,σ_1, defined (roughly) by decreasing that in a certain stage of the game it calls σ_0 or σ_1 accordingly as μ gives the value 0 or 1.

[3] An ipf "is" a set of abstract implementations in the same sense that a function "is" a set of ordered pairs – i.e. this is the way by which we represent these objects within set theory.

Rather than give the formal definition which is somewhat technical, we indicate in Fig. 2 the first few moves of the play by $\mu[\sigma_0, \sigma_1]$ for given values of μ; see also [13, 3.3]. (The picture does not indicate that if some σ_i first plays some $(a_n, \mathbf{t})$ when it is called, then from that stage on $\mu[\sigma_0, \sigma_1]$ calls the other behavior σ_{1-i} independently of the value of μ.) A merger μ is *state independent* if its values depend only on the stage of the game and not what has been played, i.e. for some $\nu: N \to \{0, 1\}$ and all sequences of states,

$$\mu(\langle s_0, \ldots, s_n \rangle) = \nu(n);$$

a merger μ is *fair* if for every σ_0, σ_1 and every infinite run of the game by $\mu[\sigma_0, \sigma_1]$, each σ_i either plays some $(a_n, \mathbf{t})$ at some stage when it is called, or is called infinitely often.

We define the (full, unfair) *merge*, the *parkmerge* and the *fairmerge* operations by the equations

$$\mathit{merge}(x, y) = \{\mu[p(0), q(0)] \mid p: N \to x, q: N \to y, \mu \text{ any merger}\},$$

$$\mathit{parkmerge}(x, y) = \{\mu[p(0), q(0)] \mid p: N \to x, q: N \to y, \mu \text{ any fair, state-independent merger}\},$$

$$\mathit{fairmerge}(x, y) = \{\mu[p(0), q(0)] \mid p: N \to x, q: N \to y, \mu \text{ any fair merger}\}.$$

Of course, we read these equations as definitions of ipfs (sets of abstract implementations), so that (for example) *merge* is the ipf generated by all

$$F_\mu(p, q) = \mu[p(0), q(0)] \quad (\mu \text{ any merger}),$$

and its abstract implementations are precisely all functions of the form

$$F_{\mu,k,l}(p, q) = \mu[p(k), q(l)] \quad (k, l \in N, \mu \text{ any merger}).$$

We use the name *parkmerge* for Park's *fairmerge*, his terminology being more appropriate in our context for the full, state-dependent fair merge operation.

In concrete modelings of concurrency closer to applications, the state structure is taken to be quite specific (typically determined by variables, stacks and buffers), and it is possible to define a large variety of "fair merge operations", see [6, 8]. It should be clear from this discussion that these operations can all be "represented faithfully" by ipfs: this is what we mean when we say that *the player model supports fairness.*

	I	s_0	s_1	s_2	s_3	s_4
μ:		1	0	1	1	0
$\mu[\sigma_0, \sigma_1]$:	II	$\sigma_1(s_0)$	$\sigma_0(s_1)$	$\sigma_1(s_0, s_2)$	$\sigma_1(s_0, s_2, s_3)$	$\sigma_0(s_1, s_4)$

Fig. 2. Action of binary merger.

1.9. The player model

We represent *processes* over a state structure $\mathscr{S}$ by *players* and *process functions* by *ipfs*, and to keep the language less stilted we will use the more familiar terms from now on:

process = player, process transformation = ipf,

except where we need to emphasize the special properties of the player model. A *process structure* for $\mathrm{FLR}_0(\tau)$ is a pair $\mathfrak{A}=(\mathscr{S},\mathscr{I})$, where the interpretation $\mathscr{I}$ assigns ipfs to the function symbols in τ.

The most substantial contribution of [13, Section 8] is a method for solving recursion equations of the form

$$x=f(x), \tag{*}$$

$$y=g(x,y), \tag{**}$$

where f and g are given process functions; the solution of (*) is a process, that of (**) a process function, and the method also solves similar systems of mutual recursion. Using this *ipf recursion* to interpret the rec construct and a a natural notion of *ipf composition*, we assign to each $\mathrm{FLR}_0(\tau)$ expression E and each list of n variables $\boldsymbol{x}$ which includes all the free variables of E, a process function f_E; closed expressions are assigned processes. We call E and M *process equivalent* if $f_E=f_M$ on every process structure.

The main result of [13, Sections 8–10] is that for FLR_0 expressions, *process equivalence coincides with behavior equivalence*, and hence also with procedure equivalence. In effect, with this modeling of interaction and concurrency, *the logic of concurrent recursion is the same as the logic of deterministic, interactive recursion, or even deterministic, non-interactive recursion*:[4] this is what we mean when we say that *the player model supports full recursion.*

1.10. Other modelings

The main – and characteristic – feature of the player model is that it supports both fairness and full recursion. Modelings like Hoare's CSP and Milner's CCS, whose primary semantics are operational do not support fairness, and the same is true of axiomatic and algebraic approaches; where fairness is included, on the other hand, as in Manna and Pnueli [8], full recursion is weakened to iteration ("while" looping),

[4] In fact the FLR_0-identities which are valid in process structures are exactly those which are valid in all domain structures $(D,\mathscr{I})$, where D is a dcpo, $\mathscr{I}$ interprets the function symbols by continuous functions on D and the recursion constructs is interpreted by the taking of least-fixed-points. This is a joint result with Tonny Hurkens, based on a simple axiomatization of the *logic of recursive equations*, which also proves the decidability and substantial robustness of this class of identities and relates this work to earlier results on recursive program schemes, trees and networks, see [3].

and Park [15] already gives a simple model which supports both fairness and iteration. Many models have also been proposed which do not support either fairness or full recursion, e.g. the "metric space" models of de Backer and Zucker [4].[5]

In the joint paper [14] with Glen Whitney, we generalize the construction of [13] to define a player model ipf(D) over an arbitrary dcpo D, we define an abstract notion of *powerstructure* which covers many models of concurrency and makes it possible to compare them, and we show that *if D is a profinite dcpo, then the Plotkin* [17] *powerdomain over D together with the set-monotone continuous functions on it is a quotient of a substructure of* ipf(D). Whitney has obtained similar "embedding" results for the Hoare and Smyth powerdomains [22].

1.11. Intensionality in the player model

The fact that a player is not determined by its traces in *1.6* exhibits some intensionality in our modeling of processes. This is quite common in many modelings of concurrency, e.g. both the strong and the weak bisimulation models of Milner's CCS [9] yield a notion of process (agent) which is not determined by its traces. What may be peculiar to the player model, is that process transformations are also modeled by intensional functions which are not completely determined by their values. It appears that this is necessary,[6] but it also adds some unexpected "expressibility" to the model. The following example may help illustrate some of the subtleties involved.

Suppose that the structure $\mathfrak{A}=(\mathscr{S},\mathscr{F})$ has a countable number of states which we identify with the integers, and consider the following two (infinitary) behavior functions F and G on $\mathfrak{A}$:

$$F(p)(\langle s_0,\ldots,s_n\rangle)\simeq\begin{cases}(skip,\partial), & \text{if } n=0,\\ p(0)(\langle s_1,\ldots,s_n\rangle) & \text{otherwise,}\end{cases}$$

$$G(p)(\langle s_0,\ldots,s_n\rangle)\simeq\begin{cases}(skip,\partial), & \text{if } n=0,\\ p(s_0)(\langle s_1,\ldots,s_n\rangle) & \text{otherwise.}\end{cases}$$

Let $f=[F]$ and $g=[G]$ be the process functions generated by F and G. It is quite obvious that these are extensionally equal, in factor for all processes x,

$$g(x)=f(x)=skip\,;x.$$

---

[5] "Guarded" recursion in the de Bakker and Zucker model obeys the laws of least-fixed-point recursion [21], but this is apparently not true for unguarded (continuous) recursion. The suggestion in [4, 3.5] that (in effect) every recursion be made guarded "by definition" is not serious: in our notation, it assigns to $x=a\,;x$ the solution $\langle skip, a, skip, a\cdots\rangle$, which does not satisfy the equation it is supposed to solve!

[6] In his Ph.D. thesis [22], Whitney has answered Question [13, 9.7] by constructing an example of two extensionally equal, unary ipfs with distinct ipf fixed points; he has constructed an extensional model which supports fairness and satisfies the "minimum conditions" for a model of concurrency listed in of [13, 6.1] – and quite a bit more; and he has shown that there is no extensional model which further satisfies some simple, natural properties of the player model.

Viewed intensionally as sets of implementations, however, $f \subset g$ but $f \neq g$, because $G(p)(\langle s_0, s_1 \rangle)$ depends on s_0 while $F(p)$ does not, and it is easy to see that "reducibility preserves state independence". Now what is the difference between the (intuitively understood) "process transformations" modeled by these two distinct mathematical objects? It appears that the best we can describe them is as follows:

$f(x)$: *Given x, call some behavior* $\sigma \in x$ *and then: to the first state* s_0 *respond by skip and after that follow* σ.

$g(x)$: *Given x: to the first state* s_0 *respond by skip, then call some behavior* $\sigma \in x$ *and after that follow* σ.

We have used "infinitary" behavior functions to capture the fact that implementations may "get access" to distinct behaviors on different "calls" to nondeterministic arguments, and we have imposed closure under reducibility to insure that process functions treat their arguments as *sets* rather than *sequences* of behaviors. The example makes it clear, however, that the modeling forces the incorporation of some aspects of the "timing" of "calls" (and their consequent, possible dependence on the state) into the modeling. It is not clear now just what role this distinction $f \neq g$ plays in the modeling, but we will see in Section 3 that it is quite important.

2. Definitions and results

Definition 2.1. A process function f on a structure $\mathfrak{A}$ is *computable*, if it is definable on $\mathfrak{A}$ by some expression E of FLR_0, i.e. $f = f_E$ in the sense of the process semantics of $\mathfrak{A}$ to which we alluded above.

Since we are interested in characterizing the computability of "true" (nondeterministic) process functions like the fair merge, we will factor out the mundanely computable behavior functions by means of the following definition.

Definition 2.2. A behavior structure $\mathfrak{A}$ as in (4) is *manageable, complete* if there exist fixed, nonrepetitive enumerations of its (countably) infinite set of states and (possibly finite) set of acts

$$\mathsf{States} = \{S_0, S_1, \ldots\}, \quad \mathsf{Acts} = \{A_0, A_1, \ldots\},$$

so that the following conditions hold.

(1) There is a fixed recursive partial function $exec : N \times N \to N$ such that $exec(i, j)$ is defined when A_j is an act and

$$S_i A_j \simeq S_{exec(i,j)}.$$

(2) For each state s, there are infinitely many distinct states which are accessible from s.

(3) Every (classical) recursive behavior function is computable in $\mathfrak{A}$.

The first two of these hypotheses certainly hold in the standard example where the state is a finite (or effectively enumerable) store of variables. They allow us to "identify" (code) states and acts with integers, but also (using standard recursive codings) to identify finite sequences of states with integers, behaviors with (special) partial functions on N to N, etc. Thus, we can talk about *recursive behavior functions* in the third hypothesis, meaning recursive functionals which take partial functions as arguments and values, and these certainly include act execution, sequential execution, conditions based on recursive conditions on the state, etc. Precise definitions will be given in Section 3.[7]

We will characterize computability in *expanded structures* of the form $(\mathfrak{A}, \mathscr{F}^*)$, obtained by adding to a manageable complete behavior structure $\mathfrak{A}$ some set $\mathscr{F}^*$ of process functions, e.g. the various merges. Formally, we should write $(\mathfrak{A}^J, \mathscr{F}^*)$, where $\mathfrak{A}^J$ is the process structure obtained by replacing each behavior function F in $\mathfrak{A}$ by its liftup F^J defined in (19), but [13, Theorem 8.9] implies that the distinction is only notational.

If F is a monotone, continuous operation of the type

$$F:(N\to\mathsf{B})^n\times(N\to N)\to\mathsf{B}, \tag{21}$$

then for each $\delta\in(N\to N)$, the *δ-section* $F_\delta:(N-\mathsf{B})^n\to\mathsf{B}$ of F is the infinitary behavior function of n arguments defined by fixing δ in F,

$$F_\delta(p_1,\ldots,p_n)=F(p_1,\ldots,p_n,\delta). \tag{22}$$

Definition 2.3. A process function f is *defined recursively from a set O* of total, number-theoretic functions, if there exists a recursive function F as in (21) such that f is generated by the δ-sections of F with $\delta\in O$, i.e. $f=[\{F_\delta\mid\delta\in O\}]$.

Notice that if f is defined recursively from O via the recursive function F, then its values are given by

$$f(\boldsymbol{x})=\{F(\boldsymbol{p},\delta)\mid p_1:N\to x_1,\ldots,p_n:N\to x_n,\delta\in O\},$$

i.e. intuitively, we can compute a value of $f(\boldsymbol{x})$ if we are "given" arbitrary calls to behaviors in the process arguments and an "oracle" $\delta\in O$. The definition makes sense for functions with 0 arguments, i.e. processes: a process x is defined recursively from the set of oracles O if there is a recursive $F:(N\to N)\to\mathsf{B}$ such that

$$x=\{F(\delta)\mid\delta\in O\}. \tag{23}$$

Theorem 2.4 (Main result). *Fix a manageable, complete behavior structure $\mathfrak{A}$.*

(1) *The computable process functions of* $(\mathfrak{A}, merge)$ *are the same as the computable process functions of* $(\mathfrak{A}, or)$; *they are the same as the computable process functions of*

[7] We want to avoid technical issues of abstract recursion and language design here. It is quite routine to turn the third hypothesis into a theorem, by enriching the language with a few simple, natural constructs.

$(\mathfrak{A}, +)$; *and they are precisely the process functions defined recursively from the Cantor set* $\mathsf{C}=(N\to\{0,1\})$.

(2) *A process function* f *is computable in the expansion* $(\mathfrak{A}, parkmerge)$ *if and only if* f *is defined recursively from the full Baire space* $\mathsf{N}=(N\to N)$.

(3) *A process function* f *is computable in the expansion* $(\mathfrak{A}, fairmerge)$ *if and only if it is defined recursively from the set of (codes of) well-founded trees*

$$\mathsf{WF}=\{\delta\in\mathsf{C}\mid(\forall\alpha:N\to N)(\exists n)[\delta(\bar{\alpha}(n))=1]\}, \tag{24}$$

where $\bar{\alpha}(n)$ *is the integer code of the finite sequence* $(\alpha(0),\ldots,\alpha(n-1))$.

(4) *In each of these cases the set of computable process functions does not change if we further expand the structure by functions which are defined recursively from the specified set of oracles.*

(5) *For each of the structures in* (1)–(3), *every computable process function has a recursive abstract implementation and every computable process contains a recursive behavior.*

It is easiest to read off some of the consequences of this theorem for the case of total processes, which may be identified with subsets of the Baire space N since all their members are totally defined behaviors. Recall the logical classification of analytical subsets of Baire space by how many and what kind of quantifiers we need to define them beginning with a recursive relation R, cf. [19, 10]. In this summary table of definitions, i varies over N and Greek letters vary over the Baire space N:

$$\Pi^0_1 \quad \delta\in x \Leftrightarrow (\forall i)R(\delta,i),$$

$$\Sigma^1_1 \quad \delta\in x \Leftrightarrow (\exists\alpha)(\forall i)R(\delta,\alpha,i),$$

$$\Sigma^1_2 \quad \delta\in x \Leftrightarrow (\exists\alpha)(\forall\beta)(\exists i)R(\delta,\alpha,\beta,i).$$

The *dual classes* to these are defined by taking the negations of these forms and they are denoted by interchanging Σ and Π, e.g. the class of complements of Π^0_1 sets is Σ^0_1. It is known that each of these classes is proper, i.e. none of them is included in its dual.

We will characterize the total processes computable in the full (unfair) *merge* in terms of a natural notion of "observables" of processes.

Definition 2.5. A *direct observable* of a behavior σ is a finite sequence

$$obs=s_0,\sigma(\langle s_0\rangle),s_1,\sigma(\langle s_0,s_1\rangle),\ldots,s_n,\sigma(\langle s_0,\ldots,s_n\rangle), \tag{25}$$

which is a legal initial part of the game of interaction, i.e. all the terms of the sequence are defined and I's partial play $\langle s_0,\ldots,s_n\rangle$ satisfies the condition of accessibility. An *ideal observable* of σ is any finite set of its direct observables.

The *direct* and *ideal* observables of a process x are the direct and ideal observables of the behaviors of x.

2.6. Observability and initial nondeterminism.

The direct observables of a player x are the "finite" facts about x that we can learn by direct "experimentation" or "testing", assuming that we have full control of the state: we adopt the role of player I in the interaction game, start with some s_0, x (perhaps) responds with some (a_0, ∂), we choose some s_1, etc. If the game goes on for $n+1$ turns, we have at the end an observable of x as in (25), where σ is the behavior followed by x in this run; this is because by our understanding of "initial nondeterminism", x "chose" a fixed behavior σ at the start of this run and he will follow it until the game ends ($\sigma(\langle s_0, \ldots, s_n\rangle) = (a_n, \mathbf{t})$) or hangs ($(\sigma(\langle s_0, \ldots, s_n\rangle) = \perp$). The ideal observables may also be viewed as facts about x which can be learned by testing, but we must be allowed some questionable "backtracking" (or "undoing") moves in the game, where (playing as I) we can "change our mind", take back a state s_i and replace it by s_i' after recording x's response to s_i – while x is not allowed to switch behaviors. We will not analyze here any further the various notions of *observational equivalence* between players and their relation to the extensive literature on this notion in process theory.

Each ideal observable of a behavior σ is (essentially) the restriction $\sigma \restriction \Delta$ of σ to a finite set Δ of sequences of states which is closed under initial segments (a *tree*) and such that σ is defined on every member of Δ. For each process x and finite tree of state sequences Δ, we let

$$idobs_x(\Delta) = \{\sigma \restriction \Delta \mid \sigma \in x, (\forall \langle s_0, \ldots, s_n\rangle \in \Delta)\sigma(\langle s_0, \ldots, s_n\rangle)\downarrow\} \tag{26}$$

be the set of *ideal observables of* x *on* Δ.

Definition 2.7. A process x is *effectively dense in itself* if from (a canonical listing of) each ideal observable $\Theta = \sigma \restriction \Delta$ of x, we can effectively find a recursive behavior $\sigma' \in x$ which realizes Θ in the sense that $\Theta = \sigma' \restriction \Delta$.

Park (adapted to the present context) would call a process x *implementable* if it has a recursive behavior, so we can think of effective self-density as a very strong form of implementability.

Theorem 2.8. *Let $\mathfrak{A}$ be a fixed, manageable, complete behavior structure. A total process x is computable in $(\mathfrak{A}, merge)$ if and only if it is closed as a subset of Baire space and for each finite tree of state sequences Δ, the set of ideal observables $idobs_x(\Delta)$ is a finite set which we can effectively list from a canonical listing of Δ.*

As a subset of Baire space, each total process x computable in $(\mathfrak{A}, merge)$ is Π^0_1, compact and effectively dense in itself.

2.9. Initial vs. persistent nondeterminism. A plausible modeling of a (total) process $\tilde{x}$ is that it is a function

$$\tilde{x} : \mathsf{States}^* \to power(\mathsf{Acts} \times \{\partial, \mathbf{t}\}) \setminus \{\emptyset\} \tag{27}$$

on nonempty sequences of states to nonempty sets of acts (paired with indicators), which acts as a *multiple-valued strategy* in the interaction game: to I's move s_0, $\tilde{x}$ responds by choosing some $(a_0, w_0)\in\tilde{x}(\langle s_0\rangle)$, to I's next move s_1, $\tilde{x}$ responds by choosing some $(a_1, w_1)\in\tilde{x}(\langle s_0, s_1\rangle)$, etc. For each world strategy ω, the responses of $\tilde{x}$ form a closed set

$$w * \tilde{x} = \{a_0, a_1, \ldots, | (\forall n)(\exists w_n)[(a_n, w_n)\in\tilde{x}(\langle s_0, \ldots, s_n\rangle)]\}, \tag{28}$$

the appropriate analog of $\omega * \sigma$ defined for a single-valued σ in (6). The (stream) members of this $\omega * \tilde{x}$ are the (observable) responses of $\tilde{x}$ to ω. This is the "persistent" variety of nondeterminism, and versions of it occur in several models of concurrency, sometimes with the added restriction that $\tilde{x}(\langle s_0, \ldots, s_n\rangle)$ is always a finite set; it is different from Park's "initial nondeterminism" we have adopted, by which a player x chooses at the beginning of the game a single (deterministic) strategy among those available to him, and then sticks with it no matter what. The persistent nondeterminism picture of a process is quite attractive, but it cannot accomodate fairness, because if $x = \mathit{fairmerge}(\bar{0}, \bar{1})$ (for example), then the set $\omega * x$ is not closed. For structures with only the full (unfair) merge to which Theorem 2.8 applies, however, if a player x is computable, then the set

$$\omega * x = \{\omega * \sigma \mid \sigma\in x\} \tag{29}$$

is closed, in fact compact. Second, each computable player x determines a computable, multiple-valued strategy

$$\tilde{x}(\langle s_0, \ldots, s_n\rangle) = \{(a_n, w_n) \mid (\exists u)[u * \langle (a_n, w_n)\rangle \in \mathit{idobs}_x(\Delta)\},$$

where $\Delta = \{\langle s_0\rangle, \langle s_0, s_1\rangle, \ldots, \langle s_0, \ldots, s_n\rangle\}$ is the tree of sequence states determined by $\langle s_0, \ldots, s_n\rangle$. Finally,

$$\omega * \tilde{x} = \omega * x;$$

in fact $\tilde{x}$ determines x completely. Thus, in the absence of fairness, our picture of initial nondeterminism is reconciled with the persistent nondeterminism picture, at least for the computable players in which we are ultimately interested.

We do not have a complete characterization of the total, computable processes for the more complex merges, in terms of classical notions, but we can say something about them.

Theorem 2.10. *Let $\mathfrak{A}$ be a fixed manageable, complete behavior structure.*

(1) *If a total process x is computable in the expansion $(\mathfrak{A}, \mathit{parkmerge})$, then x is effectively dense in itself and for each finite tree of states Δ, the set of ideal observables $\mathit{idobs}_x(\Delta)$ is recursively enumerable; as a subset of Baire space, x is Σ^1_1, possibly in $\Sigma^1_1\setminus\Pi^1_1$.*

(2) *If a total process x is computable in the expansion $(\mathfrak{A}, \mathit{fairmerge})$, then x is effectively dense in itself and for each finite tree of states Δ the set of its ideal observables $\mathit{idobs}_x(\Delta)$ is recursively enumerable; as a subset of Baire space x is Σ^1_2, possibly in $\Sigma^1_2\setminus\Pi^1_2$.*

The upshot of these results is that total processes which are computable in either of the fair merges are very effective in the "loose" sense of Park; at the same time, they can be terribly complex when we view them "extensionally" (as subsets of Baire space), even proper Σ^1_2 sets in the case of the full, state-dependent *fairmerge*.

3. Proof of the main result

Let

$$\mathsf{PF}=(N\rightharpoonup N), \qquad \mathsf{N}=(N\to N)$$

be the sets of all partial and total functions on N to N, respectively. We will assume as given the notion of a (partial) *recursive functional* with partial function and integer arguments,

$$F:\mathsf{PF}^n\times N^k\rightharpoonup N,$$

and the basic properties satisfied by that notion.[8] A functional $F:X\to\mathsf{PF}$ with values in PF is recursive if there is a recursive (partial) functional $F^*:X\times N\rightharpoonup N$ such that

$$F(x)=\lambda(n)F^*(x,n).$$

It is sometimes important that a recursive functional takes total (number-theoretic) functions to total functions; we will abbreviate this by writing

$$F:\mathsf{N}^n\times N^k\to\mathsf{N}.$$

In general, the notation $F:X\to W$ means that F (which may have been defined as a partial function on some superset of X) is in fact totally defined on X and takes values in W.

Fixing some natural recursive isomorphism of the set of infinite sequences of partial functions $(N\to\mathsf{PF})$ with PF, we will also apply recursive functionals to arguments in $(N\to\mathsf{PF})$. We also use the correspondence $i\mapsto S_i$ and $j\mapsto A_j$ supplied with a manageable, complete structure $\mathfrak{A}$ to identify the set of states with N and the set of acts with

[8] Actually, there are at least three competing notions in the literature, so that (for example) the partial functional

$$F(\alpha,\beta)\simeq \textit{if } (\alpha(0)\downarrow \textit{ or } \beta(0)\downarrow) \textit{ then } 1 \textit{ else } \perp$$

is recursive by one of them but not the others. We will only use properties of recursive functionals which are true for all three notions, but for definiteness we adopt the most natural definition via Turning machines with oracle calls. A partial functional $F:\mathsf{PF}^n\times N^l\rightharpoonup N$ is recursive, if there is a deterministic Turning machine M equipped with an extra *oracle tape* and special states O_i, for $i=1,\ldots,n$, which computes $F(\alpha_1,\ldots,\alpha_n,j_1,\ldots,j_l)$ for all unary partial function arguments $\alpha_1,\ldots,\alpha_n$ in the usual way, as if it were a partial function of its integer arguments only, but with the following rule for the computation in the oracle states: if M is in state O_i and the oracle tape has the number k on it (however we code numbers, unary or binary), then the computation stops if $\alpha_i(k)$ is undefined or proceeds with k replaced by $\alpha_i(k)$ on the oracle tape if $\alpha_i(k)$ is defined.

either N or a finite initial segment of N. From that, the set States* of nonempty sequences of states is also identified with N, by some canonical, recursive coding of sequences. If we further identify the indices ∂ and **t** with 0 and 1, respectively, and then $N\times\{0,1\}$ with N using another recursive bijection, we can code the behaviors of $\mathfrak{A}$ with (suitably restricted) partial functions $\sigma: N\rightharpoonup N$, i.e.

$$\mathsf{B}\subseteq\mathsf{PF}.$$

Finally, infinite sequences of behaviors will also be viewed as partial functions on N to N,

$$(N\to\mathsf{B})\subseteq\mathsf{PF}.$$

We will also use in the proofs several convenient notational conventions from [13], including

$$\bar{s}(n)=\langle s_0,\ldots,s_n\rangle.$$

Finite sequences of integers will often be confused with their codes, and then concatenation is understood appropriately. We will sometimes indicate whether an object or one of its various codes is used, but in most cases it is clear from the context what is meant and the introduction of explicit, coding functions would add little.

Suppose now that $O\subseteq\mathsf{N}$ is a set of total, number-theoretic functions which we take as the "oracles" and suppose f is a binary (for example) process function which is defined recursively from O via a recursive functional F as in Definition 2.3. In our intensional approach we identify f with the collection of all its abstract implementations, which (by closure under reducibility) are exactly all behavior functions of the form

$$F_{\pi,\rho,\delta}(p,q)=F(p^{\pi},q^{\rho},\delta), \tag{30}$$

with $\pi,\rho: N\to N$ and $\delta\in O$. We will abbreviate this situation by writing

$$f(x,y)=\{F(p^{\pi},q^{\rho},\delta)\mid \pi,\rho: N\to N,\delta\in O\}. \tag{31}$$

The main result of the paper characterizes the process functions computable in various expansions $(\mathfrak{A}^{J},\mathscr{F}^{*})$ of a manageable, complete structure $\mathfrak{A}$ as precisely the process functions which are representable by (31) with a suitably chosen O.

We will need a lemma about the peculiar behavior function G already discussed in Section 1.

Lemma 3.1. *If $\mathfrak{A}$ is manageable, complete and g is the process function generated by the recursive behavior function*

$$G(p)=\lambda(s_0,\ldots,s_n)\begin{cases}(skip,\partial) & \text{if } n=0,\\ p(s_0)(\langle s_0,\ldots,s_n\rangle) & \text{otherwise,}\end{cases} \tag{32}$$

then g is computable in $\mathfrak{A}^{J}$.

Proof. Here, of course, the expression $p(s_0)$ makes sense because we are identifying the states of the structure with the integers. Notice that we cannot get a trivial proof by showing that g is the lift-up G'_1 of some recursive behavior function on $\mathfrak{A}$, among the $\mathfrak{A}^J$ given by the hypothesis that $\mathfrak{A}$ is manageable, complete. This is because every abstract implementation of such a G'_1 satisfies

$$G'(p)=G_1(p(i_0))$$

with a fixed i_0, easily, so G cannot be reducible to such a G'. It is also not hard to check that g is not a composition of process functions which are lift-ups of behavior functions, which means that we must use process recursion in the proof.

Consider first the behavior function

$$H(\sigma,\tau)=\lambda(\langle s_0,\ldots,s_n\rangle)\begin{cases}(skip,\partial), & \text{if } n=0,\\ \sigma(\langle s_0-1,s_1,\ldots,s_n\rangle) & \text{if } n>0 \ \&\ s_0>0,\\ \tau(\langle s_0,\ldots,s_n\rangle) & \text{if } n>0 \ \&\ s_0=0.\end{cases}$$

This is recursive, hence computable in $\mathfrak{A}$, so let $h=H^J$ be the process function which is the lift-up of H in $\mathfrak{A}^J$ and let $y(x)$ be defined by the ipf recursion

$$y(x)=h(y(x),x).$$

We will show that y is precisely the process function g generated by G in (32). The argument is basically a direct computation, but we include it primarily as an example of how these computations go.

Notice first that the typical abstract implementation of h is of the form

$$H'(q,p)=H(q(i_0),p(j_0)) \tag{33}$$

with arbitrary i_0,j_0. By the definition of recursion, each abstract implementation of y is defined by a system $\{H_u \mid u\in N^*\}$ of such abstract implementations of h, specifically by setting

$$Y(p)=\sigma_\emptyset, \tag{34}$$

where (keeping p fixed) the system $\{\sigma_u \mid u\in N^*\}$ of behaviors is defined by the simultaneous recursion

$$\sigma_u=H_u(\lambda(i)\sigma_{u*\langle i\rangle},p).$$

Fix the particular system of representations of h,

$$H_u(q,p)=H(q(0),p(lh(u))),$$

where $lh(u)$ is the length of the sequence u. Now the behaviors

$$\sigma_u=\sigma^{lh(u)}$$

depend only on the length of u and they are determined by the recursion

$$\sigma^n=H(\sigma^{n+1},p(n)),$$

from which it follows quite easily that

$$\sigma^0(\langle s_0,\ldots,s_n\rangle)\simeq\begin{cases}(skip,\hat{\partial}) & \text{if } n=0,\\ p(s_0)(\langle s_0,\ldots,s_n\rangle) & \text{if } n>0,\end{cases}$$

i.e. $Y(p)=\sigma^0=G(p)$. This proves that the process function g generated by G is a subprocess function of y.

To prove the converse inclusion, consider an arbitrary implementation system for h which must be of the form

$$H_u(q,p)=H(q(\pi(u)),p(\rho(u)))$$

for suitable $\pi,\rho: N^*\to N$. The behaviors assigned to this system (for each fixed p) are determined by the recursion

$$\sigma_u=H_u(\lambda(i)\sigma_{u*\langle i\rangle},p)=H(\sigma_{u*\langle\pi(u)\rangle},p(\rho(u))), \tag{35}$$

and the abstract implementation of y determined by the system is given by (34) with these σ_u. It follows immediately from (35) and the definition of H that

$$\sigma_\emptyset(\langle s_0\rangle)\simeq(skip,\hat{\partial})\simeq G(p')(\langle s_0\rangle)$$

with any p'. For sequences of length greater than 1, check again from (35) and the definition of H that

$$\sigma_u(\langle 0\rangle * \bar{s}(n))\simeq p(\rho(u))(\bar{s}(n)),$$

$$\sigma_u(\langle i+1\rangle * \bar{s}(n))\simeq\sigma_{u*\langle\pi(u)\rangle}(\langle i\rangle * \bar{s}(n))),$$

so that by a simple induction on i, *for each i and each u there is some number $\rho^*(i,u)$ such that*

$$\sigma_u(\langle i\rangle * \bar{s}(n))\simeq p(\rho^*(i,u))(\bar{s}(n));$$

setting $u=\emptyset$ in this, we get the required reduction of $Y(p)$ to $G(p)$ via $\lambda(i)\rho^*(i,\emptyset)$. □

The main result follows from the next two simple lemmas, one for each direction of the claimed equivalence.

Lemma 3.2. *If $\mathfrak{A}^J$ is the lift-up of a manageable, complete behavior structure $\mathfrak{A}$, $F:(N\to\mathsf{B})^n\to\mathsf{B}$ is a recursive behavior function and f is the n-ary process function generated by F, then f is computable in $\mathfrak{A}^J$.*

Proof. We will prove the result for $n=1$, the general case being similar, so fix a recursive functional $F:(N\to\mathsf{B})\to\mathsf{B}$ and let f be the unary process function generated by it. The abstract implementations of f are all functions of the form

$$F^\pi(p)=F(p^\pi)=F(\lambda(i)p(\pi(i))), \tag{36}$$

where $\pi: N \to N$ is arbitrary. Set

$$F^*(\sigma) = F(\lambda(i)\lambda(\bar{s}(n))\sigma(\langle i\rangle * \bar{s}(n))),$$

so that F^* is the recursive behavior function, and it is quite easy to check (by a direct computation) that we can recover F from F^* using the G of Lemma 3.1,

$$F(p) = F^*(G(p)). \tag{37}$$

Since F^* is recursive, it is computable in $\mathfrak{A}$, so let

$$f^* = F^{*J}$$

be its lift-up to the process structure $\mathfrak{A}^J$; we will finish off the proof by varifying that *the process function f generated by F is the composition of g and f^**, i.e.

$$f(x) =_{\text{proc}} f^*(g(x)). \tag{38}$$

Notice here that we are required to prove the *intensional identity* (38), i.e. we must show that as process functions of x, the two sides of (38) have exactly the same abstract implementations, [13, 7.6]. Now f^* is generated by the single abstract implementation

$$F_1^*(q) = F^*(q(0)),$$

and by (37), $F(p) = F_1^*(\lambda(i)G(p))$, so F is an abstract implementation of $\lambda(x)f^*(g(x))$, hence $f \subseteq \lambda(x)f^*(g(x))$, since F generates f. For the other direction, the typical abstract implementation of the composition on the right-hand side of (38) is of the form

$$\begin{aligned}\lambda(p)F_1^*((\lambda(i)G^{\pi_i}(p))^\rho) &= \lambda(p)F^*(G(p)^{\pi_{\rho(0)}}) \\ &= \lambda(p)F(p^\pi), \text{ with } \pi = \pi_{\rho(0)},\end{aligned}$$

i.e. it is irreducible to F, which completes the proof. □

This lemma will be used to show that the lift-up $\mathfrak{A}^J$ of a manageable, complete structure and its expansions have a rich collection of computable process functions. To show that they do not have too many, we will use the next result, which assumes a minimal regularity property of sets of oracles.

Definition 3.3. An infinite *sequence coding* for a set of oracles $O \subseteq \mathbf{N}$ is any recursive functional

$$proj: \mathbf{N} \times N \to \mathbf{N}$$

such that

(1) $\mu \in O \Rightarrow (\forall i)[proj(\mu, i) \in O]$,

(2) $(\forall i)[\delta_i \in O] \Rightarrow (\exists \mu \in O)[(\forall i) proj(\mu, i) = \delta_i]$.

For example, if O is the full Baire space $\mathbf{N}$ or the Cantor set $\mathbf{C} = (N \to \{0, 1\})$, we can take

$$proj(\mu, i) = \lambda(n)\mu(\langle i, n\rangle).$$

If O is the set WF of well-founded trees, we can take

$$proj(\mu, i) = \lambda(u)\mu(\langle i \rangle * u),$$

where $*$ here stands for concatenation of sequence codes, since a tree is well-founded precisely when all the subtrees starting from the nodes of length 1 are well-founded. It will be convenient to use the same classical, Kleene notation for all sequence codings,

$$(\mu)_i =_{\text{def}} proj(\mu, i).$$

Lemma 3.4. *If $\emptyset \subsetneq O \subseteq \mathsf{N}$ is a set of oracles which admits an infinite sequence coding, $\mathfrak{A}$ is a process structure, and every given process function of $\mathfrak{A}$ is defined recursively from O, then every $\mathfrak{A}$-computable process function is also defined recursively from O.*

Proof. We use induction on the expression which defines the given process function, beginning with the obvious case of the projection function

$$f(x_1, \ldots, x_n) = x_i$$

on the variable x_i. This is defined recursively by the functional

$$F(\boldsymbol{p}, \delta) = p_i(0),$$

which makes no use of the argument in O. It is clearly enough to show that the class of process functions defined recursively from O is closed under composition and process recursion.

Composition. Suppose, for simplicity, that

$$f(x) = g(h(x)),$$

with g, h defined recursively from O via the recursive functionals G, H, the general case being similar. A typical abstract implementation of f is given by

$$\begin{aligned} F_1(p) &= G'(H_0(p), H_1(p), \ldots,) && (G' \in g, H_i \in h) \\ &= G'(H(p^{\pi_0}, \delta_0), H(p^{\pi_1}, \delta_1), \ldots, && (\delta_i \in O) \\ &= G(\lambda(i)H(p^{\pi_i}, \delta_i)^{\rho}, \varepsilon) && (\varepsilon \in O) \\ &= G(\lambda(i)H(p^{\pi_{\rho(i)}}, \delta_{\rho(i)}), \varepsilon). \end{aligned}$$

If we choose $\mu \in O$ (depending on the given parameters) so that

$$(\mu)_0 = \varepsilon, \qquad (\mu)_{i+1} = (\delta)_{\rho(i)},$$

we then see that the *typical abstract implementation of f satisfies*

$$F_1(p) = G(\lambda(i)H(p^{\pi_{\rho(i)}}, (\mu)_{i+1}), (\mu)_0) \tag{39}$$

with some $\mu \in O$ and some $\rho, \pi_i : N \to N$.

Define now the following recursive functional F, where we think of μ as a variable over Baire space:

$$F(p,\mu)=G(\lambda(i)H(\lambda(j)p(\langle i,j\rangle),(\mu)_{i+1}),(\mu)_0).$$

For each $\mu\in O$, $F_\mu=\lambda(p)F(p,\mu)$ is certainly an abstract implementation of f, since it is a special case of (39) with $\rho(i)=i$ and $\pi_i(j)=\langle i,j\rangle$. To verify that all abstract implementations of f are of this form, it is enough to show that with $\mu\in O$ so that (39) holds and some $\sigma: N\to N$ we have

$$F_1(p)=F(p^\sigma,\mu);$$

so take any σ which satisfies

$$\sigma(\langle i,j\rangle)=\pi_{\rho(i)}(j),$$

i.e. without caring what the values of σ are on noncodes of pairs, and compute

$$\begin{aligned}F(p^\sigma,\mu)&=G(\lambda(i)H(\lambda(j)p^\sigma(\langle i,j\rangle),(\mu)_{i+1}),(\mu)_0)\\&=G(\lambda(i)H(\lambda(j)p(\pi_{\rho(i)}(j)),(\mu)_{i+1}),(\mu)_0)\\&=F_1(p).\end{aligned}$$

Recursion. To keep the notation simple again, we will consider only the case of a single fixed point process $\bar{x}$, defined by the equation

$$x=f(x),$$

where f is assumed defined recursively from O. By the basic representation lemma 8.2 of [13], every behavior of $\bar{x}$ is given by

$$\sigma=\mathrm{rec}(i,p)[p(0),F_1(i,p)],$$

where F_1 is an arbitrary representation of f. The hypothesis is that

$$F_1(i,p)=F(p^{\pi_i},\delta_i)$$

holds, with a fixed recursive functional F and suitable $\pi_i: N\to N$ and $\delta_i\in O$. Thus, the typical behavior in $\bar{x}$ is given by

$$\sigma=\bar{p}(0),$$

where $\bar{p}$ is the least-fixed point of the fixpoint equation

$$p(i)=F(\lambda(j)p(\pi_i(j)),\delta_i). \tag{40}$$

If we put together all the π_i into one,

$$\pi(\langle i,j\rangle)=\pi_i(j),$$

then equation (40) which we must solve becomes

$$p(i)=F(\lambda(j)p(\pi(\langle i,j\rangle)),\delta_i). \tag{41}$$

Next define a function $\rho(w)$ on sequence codes of integers to integers by recursion on the length of the sequence (coded by) w,

$$\rho(\langle a\rangle)=a,$$

$$\rho(u*\langle a\rangle)=\pi(\rho(u),a\rangle),$$

and using this ρ, define the behavior $\bar{q}$ from $\bar{p}$ by

$$\bar{q}(w)=\bar{p}(\rho(w)). \tag{42}$$

We compute the equation satisfied by $\bar{q}$:

$$\begin{aligned}\bar{q}(w)&=\bar{p}(\rho(w))\\&=F(\lambda(j)\bar{p}(\pi(\langle\rho(w),j\rangle)),\delta_{\rho(w)}) && \text{by (41)}\\&=F(\lambda(j)\bar{p}(\rho(w*\langle j\rangle)),\delta_{\rho(w)}) && \text{by def. of } \rho\\&=F(\lambda(j)\bar{q}(w*\langle j\rangle),\delta_{\rho(w)}) && \text{by def. of } \bar{q}.\end{aligned}$$

Thus, $\bar{q}$ is a solution of the equation

$$q(w)=F(\lambda(j)q(w_*\langle j\rangle),\delta_{\rho(w)}), \tag{43}$$

and if $\tilde{q}$ is the least solution of this equation we have

$$\tilde{q}\leqslant\bar{q}. \tag{44}$$

We also need to show that $\bar{q}\leqslant\tilde{q}$ and and to do this we will check by induction on n that

$$\bar{q}^{(n)}\leqslant\tilde{q},$$

where

$$\bar{q}^{(n)}(w)=\bar{p}^{(n)}(\rho(w))$$

and $\bar{p}^{(n)}$ is the nth stage in the recursion which define $\bar{p}$. The basis case is similar to the induction step, so we compute for this:

$$\begin{aligned}\bar{q}^{(n+1)}(w)&=\bar{p}^{(n+1)}(\rho(w))\\&=F(\lambda(j)\bar{p}^{(n)}(\pi(\langle\rho(w),j\rangle)),\delta_{\rho(w)}) && \text{by def.}\\&=F(\lambda(j)\bar{p}^{(n)}(\rho(w*\langle j\rangle)),\delta_{\rho(w)}) && \text{by def. of } \rho\\&=F(\lambda(j)\bar{q}^{(n)}(w*\langle j\rangle),\delta_{\rho(w)}) && \text{by def. of } \bar{q}^{(n)}.\\&\leqslant F(\lambda(j)\tilde{q}(w*\langle j\rangle),\delta_{\rho(w)}) && \text{by ind. hyp.}\\&=\tilde{q}(w).\end{aligned}$$

Here the monotonicity of the functional F is used in the application of the induction hypothesis and the last step uses the fact that $\tilde{q}$ satisfies the fixed-point equation which defined it. Thus, we now have that

$$\bar{q} \text{ is the least-fixed point of (43).} \tag{45}$$

Since each $\delta_{\rho(w)} \in O$, we can find a single $\mu \in O$ such that for all w

$$(\mu)_w = \delta_{\rho(w)}, \tag{46}$$

and with this $\mu, \bar{q}$ is the least-fixed point of

$$q(w) = F(\lambda(j)q(w * \langle j \rangle), (\mu)_w). \tag{47}$$

If we now consider μ as a variable and let

$$H(w, \mu) = \bar{q}(w) \text{ relative to } \mu$$
$$= \text{the least-fixed point of (47)},$$

then this functional H is recursive, since it is the least-fixed point of a recursive equation;[9] the proof will be completed if we can show that

$$\bar{x} = \{H(\langle 0 \rangle, \mu) \mid \mu \in O\}.$$

We have already verified that the typical behavior of $\bar{x}$ is given by

$$\sigma = \bar{p}(0) = \bar{p}(\rho(\langle 0 \rangle)) \quad (\rho(\langle 0 \rangle) = 0)$$
$$= \bar{q}(\langle 0 \rangle)$$
$$= H(\langle 0 \rangle, \mu),$$

with the μ chosen above. On the other hand, we can show that for arbitrary $\mu \in O$, $H(\langle 0 \rangle, \mu)$ defines a behavior in $\bar{x}$ by choosing $\pi(\langle i, j \rangle) = \pi_i(j) = j$, so $\rho(u * \langle a \rangle) = a$ and then tracing the definitions

$$H(\langle 0 \rangle, \mu) = \bar{p}(\rho(\langle 0 \rangle)) = \bar{p}(0) \in \bar{x}. \qquad \square$$

To prove the main result using Lemmas 3.2 and 3.4, we will appeal repeatedly to the trivial fact that the "merging functional"

$$(\mu, \sigma, \tau) \mapsto \mu[\sigma, \tau] \quad (\mu : \mathsf{States}^* \to \{0, 1\}, \sigma, \tau \in \mathsf{B})$$

is recursive. Here we identify $(\mathsf{States}^* \to \{0, 1\})$ with the Cantor set $\mathsf{C} = (N \to \{0, 1\})$ by the obvious recursive isomorphism, i.e. we think of the set of all mergers as identical with C. We check parts (1)–(5) in turn.

Proof of Theorem 2.4 (1). Directly from the definitions, *merge*, *or* and $+$ are defined recursively from the Cantor set C, so that by Lemma 3.4 all the computable functions in the expansions of $\mathfrak{A}$ by *merge* or *or* are also defined recursively from C. To check the converse, identify some two distinct acts with the numbers 0 and 1, let

$$\bar{0} = \mathrm{rec}(x)[x, 0; x], \qquad \bar{1} = \mathrm{rec}(x)[x, 1; x]$$

[9] We are appealing here to the fundamental *first recursion theorem* of Kleene, which holds of all, natural notions of "recursive functional", see e.g. [19, XII].

be the (deterministic) processes with sole behaviors the indefinite iterations of 0 or 1, and let

$$c = merge(\bar{0}, \bar{1}) = \mathrm{rec}(x)[x, (0\ or\ 1); x]$$

be the (nondeterministic, state-independent) process whose behaviors are all binary sequences, so that (with our identifications, as a set), $c = \mathsf{C}$. The definition exhibits that c is computable in both expansions of $\mathfrak{A}$ by *merge* or *or*, so it is enough to check that every process function which is defined recursively from C is computable in the simpler expansion $(\mathfrak{A}^{\prime}, c)$. For example, if

$$f(x, y) = \{F(p, q, \delta) \mid p : N \to x, q : N \to y, \delta \in \mathsf{C}\}$$

with F recursive, let g be the process function defined by

$$g(x, y, z) = \{F(p, q, r(0)) \mid p : N \to x, q : N \to y, r : N \to z\}$$

and check immediately that $f(x, y) =_{\mathrm{proc}} g(x, y, c)$. The argument for $+$ is similar.

Proof of Theorem 2.4 (2). There is a well-known recursive correspondence between the fair binary sequence and N which goes as follows. If $\alpha(0) = 2n$ is even, we associate with α the binary sequence $\pi(\alpha)$ which gives $n+1$ consecutive 0's, then $\alpha(1)+1$ consecutive 1's, then $\alpha(2)+1$ consecutive 0's, etc. If $\alpha(0) = 2n+1$ is odd, we define $\pi(\alpha)$ similarly, but starting this time with $n+1$ consecutive 1's. The inverse of π is also recursive, easily. From this correspondence, it is immediate that *parkmerge* is defined recursively from the Baire space N, so that by Lemma 3.4, every process function in $(\mathfrak{A}^{\prime}, parkmerge)$ is defined recursively from N. The converse is proved as above, taking this time the constant

$$p = fairmerge(\bar{0}, \bar{1}),$$

which is computable in $(\mathfrak{A}^{\prime}, fairmerge)$ and checking using π, π^{-1} that every process function defined recursively from N can be obtained by the substitution of p in a process function which is generated by a recursive functional.

Proof of Theorem 2.4 (3). To check first that *fairmerge* is defined recursively from the set WF of well-founded trees, consider the following map which assigns to each sequence of well-founded trees

$$\boldsymbol{T} = T(0), T(1), \ldots$$

a fair merger $\mu_{\boldsymbol{T}}$. We will use $T(0)$ only to decide whether the merger should start giving 0's or 1's: say "give 0's" if $(0,0) \in T(0)$, else "give 1's". For this definition, let $\langle s_0, \ldots, s_n \rangle$ be the integer code of the sequence $\bar{s}(n) = (\bar{s}_0, \ldots, s_n)$ (which is never 0), the states being identified with integers again, suppose $T(0)$ told us to start with 0, and to I's play of $s_0, s_1, \ldots$ have the merger give 0 until the finite sequence $\bar{s}(n) \notin T(\langle 0 \rangle)$; next start using the tree $T(\langle \bar{s}(n) \rangle$ and to I's play $s_{n+1}, s_{n+2}, \ldots$ have the merger give 1 until the sequence $(s_{n+1}, s_{n+2}, \ldots, s_m) \notin T(\langle s_0, \ldots, s_n \rangle)$; and so on, using for the next stage

the tree $T(\langle s_{n+1}, \ldots, s_m \rangle)$. It is quite easy to verify that we get all fair (state dependent) mergers in this way. Moreover, with each well-founded tree T we can associate such a sequence by the obvious map

$$T_i = \{u \mid (i) * u \in T\},$$

and combining these two operations we have a map from the set of well-founded trees onto the set of all fair mergers. Moreover, the map is represented by a recursive functional (say) F on the subset WF of Baire space which codes the well-founded trees, so that we get the representation

$$\mathit{fairmerge}(x, y) = \{F(\delta)(p(0), q(0)) \mid p : N \to x, q \mid N \to y, \delta \in \mathsf{WF}\}$$

which shows that *fairmerge* is recursively defined from WF. By Lemma 3.4 then, every process function computable in $(\mathfrak{A}^J, \mathit{fairmerge})$ is recursively defined from WF.

The converse is verified exactly as above replacing c and p by

$$f_0 = \mathit{fairmerge}(\bar{0}, 1).$$

This is a process whose behaviors are precisely all binary functions on $(N \to \mathsf{States})$ (the plays of I) which take at least once the value 1, i.e. precisely the set of (codes of) well-founded trees.

The proof of the main result is now completed by noticing that Theorem 2.4 (4) is an immediate consequence of Lemma 3.4 and Theorem 2.4 (5) follows trivially from the fact that each of the oracle sets C, N and WF has a recursive member.

4. Total computable processes

Let us assume throughout this section that $\mathfrak{A}$ is a fixed, manageable, complete behavior structure. The proofs of Theorems 2.8 and 2.10 will follow from some simple lemmas, which combine the main result with a few elementary computations and applications of König's lemma. We restate here for easy reference the definition of the ideal observables of a *total* process x on a finite tree of state sequences Δ,

$$\mathit{idobs}_x(\Delta) = \{\sigma \restriction \Delta \mid \sigma \in x\},$$

where

$$\sigma \restriction \Delta = \{\langle \bar{s}(n), \sigma(\bar{s}(n)) \rangle \mid \bar{s}(n) \in \Delta\}.$$

Notice that each $\sigma \restriction \Delta$ is a finite object which can be represented by an integer, via some canonical coding.

Lemma 4.1. *If x is a total process, recursively defined from the Cantor set* C, *then x is compact (as a subset of Baire space) and for each finite tree of state sequences Δ, the set $\mathit{idobs}_x(\Delta)$ is finite and can be effectively listed from a listing of Δ.*

Proof. By definition, we know that there exists a recursive (hence continuous) function $F:\mathsf{C}\to\mathsf{B}$ such that

$$x=F[\mathsf{C}]=\{F(\delta)\mid\delta\in\mathsf{C}\}\subseteq\mathsf{N}, \tag{48}$$

so x is certainly compact. For each finite tree of state sequences Δ, set

$$G_\Delta(\delta)=F(\delta)\restriction\Delta$$

and notice that G_Δ is a continuous function, total on the Cantor set C and with discrete values (essentially) in N. By König's lemma there is a fixed integer m such that $G_\Delta(\delta)$ depends only on the first m values of δ. It follows that the image $G_\Delta[\mathsf{C}]=idobs_x(\Delta)$ is finite, we can find one such bound m by a dumb search and we then use it to give a listing of $idobs_x(\Delta)$. □

Lemma 4.2. *If x is a total process, closed as a subset of Baire space and such that for each finite tree of state sequences Δ, the set $idobs_x(\Delta)$ is finite and can be effectively listed from a listing of Δ, then x is recursively defined from the Cantor set* C.

Proof. Recall that we have identified the states with integers and for each i, let Δ_i be the *finite tree of all state sequences which have length at most i and involve only the first* $i+1$ *states.* Notice that

$$\Delta_0\subseteq\Delta_1\subseteq\cdots,$$

so for any behavior σ,

$$\sigma\restriction\Delta_0\subseteq\sigma\restriction\Delta_1\subseteq\cdots.$$

Consider now the tree $\mathscr{T}$ of all finite sequences of the form

$$(\Theta_0,\Theta_1,\ldots,\Theta_n), \tag{49}$$

so that the following conditions hold.

(1) For each $i\leqslant n$, there is some $\sigma\in x$ such that

$$\Theta_i=\sigma\restriction\Delta_i,$$

i.e. Θ_i is an ideal observable of x on Δ_i.

(2) $\Theta_0\subseteq\Theta_1\subseteq\cdots\subseteq\Theta_n$.

We can think of a coded version of $\mathscr{T}$ as a tree on integers, since its nodes are finite objects which can be listed, and as such, quite obviously from the hypothesis on x, $\mathscr{T}$ *is a recursive, finitely branching tree.* Moreover, *every sequence in* $\mathscr{T}$ *has a proper extension in* $\mathscr{T}$, because its last term is an ideal observable of some behavior of x, which is total and therefore has a "fuller" observable on the next, larger tree of state sequences. Each Θ_i which occurs in a sequences of $\mathscr{T}$ determines some partial behavior $\sigma(\Theta_i)$ defined on the sequences in Δ_i, such that

$$\Theta_i=\sigma(\Theta_i)\restriction\Delta_i$$

and $\sigma(\Theta_i)$ has a total extension in x. Moreover, for every sequence in $\mathscr{T}$ as in (49), we have

$$\sigma(\Theta_0)\subseteq\sigma(\Theta_1)\subseteq\cdots\subseteq\sigma(\Theta_n),$$

so that we can map each infinite branch

$$\boldsymbol{\Theta}=(\Theta_0\subseteq\Theta_1\subseteq\cdots)$$

of $\mathscr{T}$ onto a behavior

$$\sigma(\boldsymbol{\Theta})=\lim_n\sigma(\Theta_n).$$

The hypothesis that x is closed implies that $\sigma(\boldsymbol{\Theta})\in x$, so the construction gives an effective map from $\mathscr{T}$ onto x. The proof is completed by composing this map with some canonical, recursive surjection of the Cantor set onto the infinite branches in $\mathscr{T}$, which exists because $\mathscr{T}$ is a finitely branching, recursive tree. □

Lemma 4.3. *Every total process computable from* C *is* Π^0_1, *as a subset of* N.

Proof. The player $x=F[\mathsf{C}]$ is the continuous image of a compact set, hence compact, hence closed, and every closed subset of N is "Π^0_1 in some parameter", so we are almost there – but we need to check that the parameter is not needed, i.e. give a direct Π^0_1 definition of x. By the basic representation (48) for x we have

$$\sigma\notin x \Leftrightarrow (\forall\delta\in\mathsf{C})\underbrace{[F(\delta)\neq\sigma]};$$

for a fixed σ, the condition on δ above the brace is open and by another simple application of König's lemma and the recursiveness of F we have a semi-effective procedure for verifying that it is true of all $\delta\in\mathsf{C}$. This proves that the complement of x is Σ^0_1, so x is Π^0_1, as required. □

Lemma 4.4. *Every total process, recursively defined from* C, N *or* WF *is effectively dense in itself.*

Proof. Suppose that $x=F[O]$ with F recursive and the relevant O and let Θ be an ideal observable of x, so that for some $\delta\in O$, we have

$$(\exists\Delta)[F(\delta)\restriction\Delta=\Theta]. \tag{50}$$

This is an open condition on δ, so if satisfiable it holds for all δ (in O or outside O) which extend some finite sequence of values w, and we can certainly find such a w by a dumb search. The (trivial) key fact is that every w has recursive extensions $\delta\in O$ for the Os in which we are interested, and any such δ satisfies (50). □

With the main result, these lemmas imply immediately Theorem 2.8 and some of Theorem 2.10. The remaining claims of Theorem 2.10 are trivial consequences of the main result and known properties of the analytical classes of relations.

References

[1] M. Abadi, L. Lamport and P. Wolper, Realizable and unrealizable specifications of reactive systems, in: *Proc. 1989 ICALP*, to appear.

[2] M. Broy, A theory for nondeterminism, parallelism, communication, and concurrency, *Theoret. Comput. Sci.* **45** (1986) 1–61.

[3] B. Courcelle, Recursive applicative program schemes, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science* (Elsevier, Amsterdam, 1990), 461–492.

[4] J.W. de Bakker and J.I. Zucker, Processes and the denotational semantics of concurrency, reprinted in *Ten Years of Concurrency Semantics* (World Scientific, Singapore, 1992).

[5] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).

[6] N. Francez, *Fairness* (Springer, Berlin, 1986).

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

[8] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems* (Springer, Berlin, 1992).

[9] R. Milner, *Communication and Concurrency* (Prentice Hall, Englewood Cliffs, NJ, 1989).

[10] Y.N. Moschovakis, *Descriptive Set Theory*, Studies in Logic (North-Holland, Amsterdam, 1980).

[11] Y.N. Moschovakis, The formal language of recursion, *J. Symbol. Logic* **54** (1989) 1216–1252.

[12] Y.N. Moschovakis, Computable processes, in: *Proc. POPL Meeting*, San Francisco, 1990.

[13] Y.N. Moschovakis, A model of concurrency with fair merge and full recursion, *Inform. and Comput.* **93** (1991) 114–171.

[14] Y.N. Moschovakis and G.T. Whitney, Powerdomains, powerstructures and fairness, presented at the 1994 Ann. Conf. of the European Association for Computer Science Logic; submitted.

[15] D. Park, On the semantics of fair parallelism, in: *Proc. Copenhagen Winter School*, Lecture Notes in Computer Science, Vol. 104 (Springer, Berlin, 1980) 504–526.

[16] D. Park, The "fairness' problem and nondeterministic computing networks, in: *Foundations of Computer Science IV* (Mathematisch Centrum, Amsterdam, 1983) 33–162.

[17] G.D. Plotkin, A powerdomain construction, *SIAM J. Comput.* **5** (1976) 452–487.

[18] A. Pnueli and R. Rosner, On the synthesis of a reactive module, in: *Proc. 1989 ICALP*, to appear.

[19] H. Rogers, Jr., *Theory of Recursive Functions and Effective Computability* (McGraw-Hill, New York, 1967).

[20] M.B. Smyth, Modeling concurrency with partial orders, *J. Comput. System Sci.* **16** (1978) 23–36.

[21] V. Stoltenberg-Hansen and J.V. Tucker, Algebraic and fixed point equations over inverse limits of algebras, *Theoret. Comput. Sci.* **87** (1991) 1–24.

[22] G.T. Whitney, Recursion structures for non-determinism and concurrency, Ph.D. Thesis, Department of Mathematics, University of California, Los Angeles, 1994.