**When Is Partial Trace
Equivalence Adequate?**

Bard Bloom

91-1218
July 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Supported by NSF grant (CCR-9003441).

When Is Partial Trace Equivalence Adequate?

Bard Bloom*
Cornell University
bard@cs.cornell.edu

July 10, 1991

Abstract: Two processes are *partial trace equivalent* iff they can perform the same sequences of actions in isolation. Partial trace equivalence is perhaps the simplest possible notion of process equivalence. In general, it is too simple: it is not usually an adequate semantics. We investigate the circumstances under which it is adequate, which are surprisingly rich. We give two substantial classes of languages for which partial traces are adequate. In one class, partial trace equivalence suffices for total correctness, and operations such as true sequencing are possible; but all processes are determinate and silent moves are not possible. The other class admits indeterminacy and silent moves, but partial traces only suffice for partial correctness and true sequencing is not definable.

1 Introduction

The last decade has seen an explosion of research into simple languages for concurrency. This interest is roughly analogous to the interest in the simply-typed λ -calculus: both process algebras and simply-typed λ -calculus model elemental aspects of concurrency and higher-order programming respectively, allowing elucidation of fundamental issues without unnecessary complexity. One of the central questions of this research has been, “when are two processes equivalent?” [AV90,BK86,BM90,BIM90,BC89,BHR84,dNH84,GG89,vGW89,Hen83,Mil83,PP90,Wal90,KLP90,Gla90]. Answers to

*Supported by NSF grant (CCR-9003441)

this question inform most aspects of the theory and practice of concurrency, from language design to optimizing compilers.

Many notions of process equivalence arise as notions of *congruence*. Choose a set of aspects of process behavior which are to be considered important; this is the set of *observations*. Describe the set of ways in which processes will be used; this is the set of *contexts*. Two processes are congruent (with respect to the given observations and contexts) if the same observations obtain in all contexts. If the observations and contexts are the right ones for the situation, the congruence is precisely the right notion of process equivalence.

Many equivalences are chosen by taking a simple programming language and notion of observation, and characterizing the appropriate notions of congruence, generally characterizing it in a somewhat language-independent way. This is the methodology of [BIM90.BHR84,dNH84,GV89].

In this paper, we work in the other direction. We start with perhaps the simplest semantic model, the *partial trace model*, and describe a large class of languages for which this model is appropriate. This study is of a larger research program linking semantic models and classes of programming languages.

The partial trace model (in one form or another) has been rediscovered by almost every researcher working on concurrency in any of its aspects. It is simple and appealing. A process is a construct capable of performing *actions*; two processes are considered identical iff they can perform the same finite sequences of actions.

It is well-known that partial traces are not adequate for most languages for concurrency; we remind the reader of the counterexamples in Section 1.1. However, the partial trace model is one of the most straightforward and elementary models of concurrency, easily explained and used. It would be useful to be able to design languages for which partial traces are adequate.

Definition 1.1 *A language $\mathcal{L}$ is partial-trace believing if partial traces give an adequate semantics; that is, whenever P and Q have the same partial traces, so do $C[P]$ and $C[Q]$ for all $\mathcal{L}$ -contexts $C[\cdot]$.*

In this paper, we give sufficient conditions on the adequacy of partial traces: two large classes of languages in which partial traces give adequate semantics. The first class, the *partial trace determinate* languages, places substantial restrictions on both processes and operations, resulting in a class of languages for which partial traces give *all* the behavior of processes, and

may be used to specify total correctness. The second class, the *straight tyft* languages, is more general and thus weaker; partial traces are good enough to specify partial correctness, but not total correctness, in straight tyft languages.

To investigate this area, we need some way to describe classes of languages. Most calculi for concurrency discussed above are presented as *Structured Operational Semantics* (SOS) [Plo81,Blo89,dS85,GV89], in which the behavior of composite process terms is defined in terms of the behavior of their subterms. We describe classes of languages in terms of their SOS definitions. This has several advantages:

1. It is easy to test the conditions, by inspection of the rules of the language.
2. It is easy to define a partial-trace believing language, simply by working inside one of the classes provided.
3. It is frequently desirable to extend languages. In general, extending a language requires changes in the notion of process equivalence the language uses. However, extensions which remain inside the classes of this paper will not require changes in the notion of process equivalence: partial traces (for the work in this paper) will remain adequate no matter which operations are added.

In fact, partial traces will be *fully abstract* (observing partial traces) for all languages in the classes: if two processes behave identically in all contexts, then they have the same set of partial traces. Full abstraction is often a difficult theorem, and many languages (including CCS) are not fully abstract for their preferred models. In this case, though, full abstraction is a fairly trivial result given adequacy. The meaning of a process is its set of partial traces. If two processes agree in all contexts, then they agree (viz. have the same traces) in the identity context: in particular they have the same meaning.¹

1.1 Summary of Results

There are two fundamental reasons why partial traces do not provide good semantics: (1) distinguishable indeterminate processes may have the same

¹This trivial agreement of adequacy and full abstraction will hold in any semantic model in which the meaning of a term is simply the set of observations about the term.

partial traces, and (2) partial traces are not in general capable of detecting total correctness. We use CCS notation for discussing these results.

1.1.1 Determinate Processes

The processes ab and $a + ab$ have the same set of partial traces: $\{\langle\rangle, a, ab\}$. Many process calculi have *sequencing operators*. Ideally, one would like to have $P ; Q$ be a process which first executes P , and then executes Q .² The processes ab and $a + ab$ behave differently under sequencing. $ab ; a$ has partial traces $\{\langle\rangle, a, ab, aba\}$, while $(a + ab) ; a$ has partial traces $\{\langle\rangle, a, aa, ab, aba\}$.

Engelfriet [Eng85] and Vaandrager [Vaa88] have shown that, for *determinate* processes, most forms of process equivalence coincide, and in particular they are equal to partial trace equivalence. In Section 3, we provide a partial converse to this.

Result 1: In Section 3, we show that, in any language in which partial traces are adequate and a sequencing operator is present, all processes are determinate. Conversely, we show that in any GSOS language in which all processes are determinate, partial traces are adequate – and indeed adequate for total correctness. We develop the theory of *nicely determinate* languages, in which all processes are determinate and in which this theory applies.

Result 1 collapses completely in the presence of hidden moves.

1.1.2 Partial Correctness

The processes $a(b + c)$ and $ab + ac$ have the same set of partial traces: $\{\langle\rangle, a, ab, ac\}$. They are considered different because of the following fact. Let $C[X] = X \setminus c$; that is, $C[P]$ is the process P forbidden to do c -moves. $C[a(b + c)]$ is able to do an a , and then do a b . However, $C[ab + ac]$ makes an internal choice on its first move. It might be able to do an a followed by a b ; or it might simply be able to do an a and then unable to continue. The designers of CCS are concerned with total correctness and so consider these processes distinct.

It is worth noting that $C[a(b + c)]$ and $C[ab + ac]$ have the same set of *partial* traces. Indeed, $a(b + c)$ and $ab + ac$ are CCS congruent when observing partial traces [dNH84]: if one is only concerned about the *possibility* of getting a result rather than the *necessity*, partial traces are adequate.

²For various reasons, most languages have a different sequencing operator; this is discussed more in [Hoa78] and Section 3.3.2.

Result 2: In Section 4, we generalize this result and show that all languages defined by *straight tyft rules* — an extensive class of rules, of which CCS is a simple representative — are partial-trace believing.

Result 2 can be extended in a straightforward way to the presence of hidden moves. Indeed, the main difficulty in the extension is defining what a structured operational semantics for hidden moves may be.

1.2 Discussion

Result 1 concerns determinate processes, but “determinate” is a somewhat tricky term. See [Mil89] for a detailed discussion of determinacy. The process $a + b$, which can choose to do either an a or a b , is determinate. Determinate processes are allowed to make choices; they are merely required to announce their choices to the environment. That is, $ab + ac$ is not determinate; its first action a does not give enough information to tell whether b or c will occur next.

The main uses of nondeterminacy in process algebra are:

1. Expressing concurrency. In many settings, parallel composition of atomic actions, $a|b$, is equivalent to sequential composition in either order: $ab + ba$. Most parallel composition operations used in practice preserve determinacy if used on suitably non-overlapping processes, though most can introduce nondeterminacy in general.
2. Expressing receptivity. For example, let y and n be the atomic actions of pressing the y-key and n-key on the keyboard. Then the process $y.P_y + n.P_n$ accepts a y or n input, and acts accordingly. In most situations, the behavior of the process is uniquely determined by the input, and so most receptive uses of nondeterminacy are actually determinate.
3. Specifying processes. If P_1 and P_2 are both acceptable behaviors of a process after the y key is pressed, then the process could be specified as $y.P_1 + y.P_2$. This use of nondeterminacy is clearly not determinate.

So, it seems likely that in many settings, requiring that *processes* be determinate is a fairly minor restriction, and so it is worthwhile investigating determinate calculi. Specification by means of processes will be more difficult in this setting. However, one of the common uses of partial trace semantics is specification of processes by specifying their set of partial traces [Udd86,Maz89]. So losing the ability to use processes as specifications is

unlikely to be a great concern. By the results of [Eng85], we know that the partial traces of determinate processes in fact determine the completed traces and most other semantic information, so specifying a determinate process up to partial traces actually gives a *total* correctness specification. We thus consider the determinacy results in this study to be of significant value.

Result 2, concerning partial correctness, is of less certain value. In most concurrent settings, deadlock is an undesirable property; if a system may deadlock, it probably should be considered wrong. The partial trace semantics of this section ignore deadlock; that is, the processes ab and $ab + a.\text{deadlock}$ have the same partial traces, and are thus not distinguished. As in the sequential setting, specifying partial traces is likely not to be enough; separate arguments for liveness will probably be required.

Many process calculi have a notion of *silent* or *internal move*, a special action τ which is regarded as unobservable. Process semantics are generally somewhat tricky in the presence of silent moves. In particular, the possibility of *silent divergence* — a process taking silent moves forever — introduces complications [Wal90]. Partial traces (ignoring τ 's) are not an adequate model of any language with sequencing, and Result 1 fails in the presence of silent moves.³ However, there are several straightforward extensions of Result 2 which apply to silent moves.

1.2.1 Enhanced Trace Models

Many of the proofs and counterexamples in this study depend on the use of partial traces, and the results would be different if we used an enhanced partial trace model. For example, some of the discussion in Section 3 uses the fact that the partial trace model does not include termination information; using completed or successful traces of some sort would make the proof invalid. The discussion of silent moves relies on the fact that partial traces do not include divergence information; adding this information would change this result as well.

This study is only concerned with the simplest case, that of partial traces which do not include extra information. Enhanced trace models are important, and will be discussed elsewhere. It seems desirable to understand the

³We consider the trace-like models of CSP and ACP to be partial trace models enhanced with termination information, and hence not pure partial trace models. Partial trace models with termination information will be discussed in a later study.

$$\frac{X \xrightarrow{a} Y}{X + X' \xrightarrow{a} Y} \qquad \frac{X' \xrightarrow{a} Y'}{X + X' \xrightarrow{a} Y'}$$

Figure 1: Rules for +

base case of unadorned partial traces – which is fairly complex in its own right – before analyzing the more elaborate cases.

2 Basic Definitions

2.1 Process Algebra

We assume familiarity with the basic notation of process algebra; see *e.g.* [BK84,BIM90,Mil83] for more details. We work with *actions*, elements of some finite set **Act**. a, b, c, d denote actions; s, t are strings of actions. $\langle \rangle$ is the empty string.

Processes are represented as terms in some algebra. The basic notion of process behavior is $P \xrightarrow{a} P'$, meaning that the process P can perform the action a and thereafter behave like P' . We extend this notation to strings: $P \xrightarrow{\langle \rangle} P$, and $P \xrightarrow{a} P' \xrightarrow{s} P''$ implies $P \xrightarrow{as} P''$. We write $P \xrightarrow{s}$ if $\exists P'. P \xrightarrow{s} P'$, and $P \not\xrightarrow{s}$ otherwise. Finally, $P \nrightarrow$ iff for all a , $P \not\xrightarrow{a}$.

Three of the basic operations, present in some form in most process algebras, are the null process $\mathbf{0}$, prefixing $a(\cdot)$ and nondeterministic choice $+$. The null process is incapable of taking any action, and consequently has no rules. $aP \xrightarrow{a} P$ for all actions a and processes P . $P + Q$ is defined by the rules in Figure 1. The intent of these rules is that, whenever a transition is known to be possible for P or Q , that transition is possible for $P + Q$. For example, we have $a(b\mathbf{0} + c\mathbf{0}) \xrightarrow{a} b + c \xrightarrow{b} \mathbf{0} \xrightarrow{c} \mathbf{0}$. By convention, we write a for $a\mathbf{0}$ in process terms.

Process algebras are, first of all, algebras; we omit definitions of “term,” “arity,” and other standard terms. Almost all operators in almost all process algebras are defined by rules which resemble those in Figure 1 to some degree. For our treatment of classes of languages, it is useful to present this form in some generality. In [BIM88,Blo89,BIM90], we presented a useful generalization as GSOS rules; Vaandrager and Groote [GV89] gave another

generalization as *tyxt/tyft* rules. These formats are both maximal, in the sense that any each one enjoys some internal consistency properties and no simple extension has these properties. Both notions start with the concept of a *transition formula*:

Definition 2.1 A positive transition formula is a triple of two terms and an action, written $T \xrightarrow{a} T'$. A negative transition formula is a pair of a term and an action, written $T \not\xrightarrow{a}$.

In general, the terms in the transition formula will have free variables; the $X \xrightarrow{a} Y$ in Figure 1 is a typical example. The GSOS format allows both positive and negative transitions, but only for variables.

Definition 2.2 A GSOS rule ρ is a rule of the form:

$$\frac{\left\{ X_i \xrightarrow{a_{ij}} Y_{ij} \mid 1 \leq j \leq m_i \right\}_{i=1}^l, \quad \left\{ X_i \not\xrightarrow{b_{ik}} \mid 1 \leq k \leq n_i \right\}_{i=1}^l}{\alpha(X_1, \dots, X_l) \xrightarrow{c} C[\vec{X}, \vec{Y}]}$$

where all the variables are distinct, $l \geq 0$ is the arity of the operator symbol α , $m_i, n_i \geq 0$, and $C[\vec{X}, \vec{Y}]$ is a context with free variables including at most the X 's and Y 's. (It need not contain all these variables.)

It is useful to name components of rules. The operation symbol α is the principal operator of the rule, and the term $\alpha(\vec{X})$ is the source. $C[\vec{X}, \vec{Y}]$ is the target; c is the action; the formulas above the line are the antecedents; and the formula below the line is the consequent.

A GSOS language is a finite set of operators, a finite action alphabet, and a finite set of GSOS rules defining each operator.

The intent of a GSOS rule is as follows. Suppose that we are wondering whether $\alpha(\vec{P})$ is capable of taking an a -step. We look at each rule with principal operator α and action a in turn. We check each positive antecedent $X_i \xrightarrow{a_{ij}} Y_{ij}$, checking if P_i is capable of taking an a_{ij} -step for each j and if so calling the a_{ij} -children Q_{ij} . We also check the negative antecedents; if P_i is incapable of taking a b_{ik} step for each k . (It is reasonably easy to show that $P_i \not\xrightarrow{b_{ik}}$ is decidable, by structural induction.) If so, then the rule *fires* and $\alpha(\vec{P}) \xrightarrow{c} C[\vec{P}, \vec{Q}]$.

For example, let β be the operator given by the single rule⁴

$$X \xrightarrow{a} Y, X \not\xrightarrow{b} /- \beta(X) \xrightarrow{a} Y$$

⁴We occasionally write rules in-line with the following notation: $\chi_1, \dots, \chi_n /- \chi$, where each χ is a transition formula.

Then $\beta(aa + ab) \xrightarrow{a} a$. However, $\beta(aa + ab + b)$ is stopped (that is, has no behavior), as the negative antecedent of the rule for β is not satisfied. Further examples include Figure 1, (3), (3.1), (3.1) and elsewhere in this paper, as well as most rules in most papers on process algebras.

In [Blo89,BIM90] we argue in favor of the GSOS discipline. Briefly, GSOS rules seem to be a maximal class of rules such that:

1. Every GSOS language has some basic sanity properties; *e.g.*, the relation defined informally above can be defined formally; it always exists (which is not to be taken for granted, given negative antecedents), and indeed is computable and finitely branching; and it respects many of the stronger notions of process equivalence, in particular bisimulation and ready simulation.
2. It seems impossible to extend the format of the rules in any systematic way which preserves the basic sanity properties. Unlike 1, this is informal; [Blo89] gives a series of examples showing that the most natural extensions are not allowable. [GV89,BG90] give variants of the GSOS format which are consistent; though they do not have our basic sanity properties, they have some different but still reasonable ones.

Most process calculi allow recursive definitions of terms. This can be added to GSOS languages in a quite natural way. Doing so does not change most of the properties we consider (including those in this study), but it does triple the number of cases in most proofs and complicate most inductions.

Definition 2.3 *A GSOS rule is satisfiable iff there are no positive and negative antecedents $X \xrightarrow{a} Y$ and $X \not\xrightarrow{a}$ for the same variable and action.*

Note that a GSOS rule for α is satisfiable iff there is some term (involving $+$ and atomic actions) which causes it to fire.

2.2 Partial Traces and Determinacy

Definition 2.4 *We say that s is a partial trace of P if $P \xrightarrow{s}$. $\text{ptr}(P)$ is the set of all partial traces of P .*

For example, $\text{ptr}(ab + ac) = \{\langle \rangle, a, ab, ac\}$.

Definition 2.5 *P and Q are partial trace equivalent, $P \equiv_{\text{ptr}} Q$, iff $\text{ptr}(P) = \text{ptr}(Q)$. They are partial trace congruent with respect to $\mathcal{L}$, $P \equiv_{\text{ptr}}^{\mathcal{L}} Q$, iff for all $\mathcal{L}$ -contexts $C[X]$, $C[P] \equiv_{\text{ptr}} C[Q]$.*

In general, $P \equiv_{\text{ptr}} Q$ does not imply $P \equiv_{\text{ptr}}^{\mathcal{L}} Q$ for most languages $\mathcal{L}$; this paper is ultimately a study of sufficient conditions under which the two relations coincide.

If s is a string, $\text{prefixes}(s)$ is the set of all prefixes of s ; we extend this to sets of strings in the obvious way. Note that $\text{ptr}(P)$ is always *prefix-closed*; that is, $\text{prefixes}(\text{ptr}(P)) = \text{ptr}(P)$.

Half of the results in this paper will concern *determinate* processes, that is, processes whose state (up to some equivalence) can be determined from their initial state and the sequence of actions which they are known to have taken. Note that they may behave unpredictably, by choosing to output either an a or a b ; determinacy simply forbids making an unpredictable choice and keeping it hidden.

Definition 2.6 *Let $\equiv$ be a notion of process equivalence, which for this definition can be any equivalence relation between processes. P is $\equiv$ -determinate iff, whenever $P \xrightarrow{s} P'$ and $P \xrightarrow{s} P''$, then $P' \equiv P''$.*

This study will be concerned with two special cases:

Partial-trace Determinacy: P is partial-trace determinate iff whenever

$$P \begin{array}{c} \xrightarrow{s} P_1 \\ \xrightarrow{s} P_2 \end{array}, \text{ we have } \text{ptr}(P_1) = \text{ptr}(P_2).$$

Term Determinacy: P is term determinate iff whenever $P \begin{array}{c} \xrightarrow{s} P_1 \\ \xrightarrow{s} P_2 \end{array}$, we have $P_1 = P_2$; that is, they are the same term.

3 Determinacy, Sequencing, and Negative Rules

In Section 3.1, we show that, given sufficiently powerful operations (*e.g.*, sequencing), all partial-trace believing languages are determinate. Section 3.2 provides a converse: we show that a GSOS language in which all terms are determinate is partial-trace believing. Also, we define a large class of determinate GSOS languages. We chose sequencing as a fairly plausible operation defined with negative rules; many other such operations (*e.g.*, polling, priority, and broadcasting operations) work as well.

Many languages might reasonably wish to include an operation $;$, called “true sequencing.” $P ; Q$ runs P until it can run no further, and then runs Q . This is a fairly powerful operation; for example, if P deadlocks, then Q is executed.

3.1 Sequencing Implies Determinacy

It is frequently the case that, if a language uses operations defined with negative rules and partial traces are adequate, then the language must be partial trace determinate. We demonstrate this with a true sequencing operation, and afterwards briefly describe other operations that have a similar effect.

Suppose that we have a true sequencing operation “ $;$ ”. True sequencing may be defined by the rules

$$\frac{P \xrightarrow{a} P'}{P ; Q \xrightarrow{a} P' ; Q'} \quad \frac{P \nrightarrow, Q \xrightarrow{a} Q'}{P ; Q \xrightarrow{a} Q'}$$

Assume also we have some standard form of parallel composition. Our proof is fairly insensitive to precisely which form we take; we choose the LOTOS-style [BB87] operation $P|_L Q$, which runs P and Q in parallel. L is a sets of actions. The intent is that P and Q must synchronize on actions in L , and may not synchronize on other actions.

$$\frac{P \xrightarrow{a} P', \quad a \notin L}{P|_L Q \xrightarrow{a} P'|_L Q, \quad Q|_L P \xrightarrow{a} Q|_L P'} \quad \frac{P \xrightarrow{a} P', \quad Q \xrightarrow{a} Q', \quad a \in L}{P|_L Q \xrightarrow{a} P'|_L Q'}$$

This definition was chosen for simplicity of exposition. Most process calculi have a form of parallel composition which suffices for Theorem 3.1. For example, in CCS a very similar proof with a term of the form $(P|Q)\backslash L$ works.

Theorem 3.1 *In any partial-trace believing language $\mathcal{L}$ including action prefixing, at least two atomic actions, and the sequencing and parallel composition operations, all processes must be partial-trace determinate.*

Proof: Suppose that there is some process Q which is not partial-trace determinate. That is, we have $Q \xrightarrow{a} Q_1$, where $Q_1 \not\equiv_{\text{ptr}} Q_2$. Suppose, without loss of generality, that there is some string $s \in \text{ptr}(Q_1) - \text{ptr}(Q_2)$.

Choose s to be a minimum length string with this property; that is, $s = rc$, and $r \in \text{ptr}(Q_1) \cap \text{ptr}(Q_2)$. Let $r_1 \dots r_n$ be the characters of r , in order.

As atomic actions and sequencing are present in the language, $P = arc = a.r_1 \dots r_n.c.\mathbf{0}$ is a definable process. Let $\dot{Q} = (Q|_{\text{Act}}P)$. Notice that $\dot{Q}$ is a finite process, with $\text{ptr}(\dot{Q}) = \text{prefixes}(\{arc\}) = \text{ptr}(P)$.

Choose an action $d \neq c$.

$$C[X] = X ; d \quad (1)$$

Now we see that

$$C[\dot{Q}] \xrightarrow{a} (Q_2|_{\text{Act}}r_1; \dots; r_n; c) ; d \xrightarrow{r} (Q'_2|_{\text{Act}}c) ; d \quad (2)$$

and as $rc \notin \text{ptr}(Q_2)$, we have $(Q'_2|_{\text{Act}}c) \nrightarrow$ and thus $C[\dot{Q}] \xrightarrow{ar} (Q'_2|_{\text{Act}}c) ; d \xrightarrow{d} \mathbf{0}$, and we have shown that $ard \in \text{ptr}(C[\dot{Q}])$.

However, by the definition of P , we have precisely one computation $P \xrightarrow{ar} P' = c.\mathbf{0}$, and as $arc \in \text{ptr}(P)$, we have $P' \xrightarrow{c} \mathbf{0}$. Hence $C[P] \xrightarrow{ar} P' ; d$. P' is not terminated, and hence the ar trace of $C[P]$ can only be extended $C[P] \xrightarrow{arc} (\mathbf{0}) ; d$. In particular, we have $ard \notin \text{ptr}(C[P])$.

Thus, $\mathcal{L}$ is not partial-trace believing. $\triangleleft$

A variety of operations allow this or a similar proof to work. Perhaps the best example is a *polling* operation, which allows processes to run in parallel, but to detect whether their partner is willing to perform certain actions. Frits Vaandrager points out that a variety of combinations of parallel, priority, and renaming operators will allow this theorem to work. It is an open problem to determine precisely which operations imply partial trace determinism. We conjecture that any language containing a perpetual operation defined with negative rules – that is, one which repeatedly tests its argument, allowing it to run in some cases, and applies a negative rule to it – can be extended by some positive GSOS operations to give a language in which Theorem 3.1 holds.

3.2 Structured Operational Semantics of Determinate Languages

A sort of converse of Theorem 3.1 follows straightforwardly from [Eng85].

Theorem 3.2 *Let $\mathcal{L}$ be a GSOS language in which all processes are partial trace determinate. Then $\mathcal{L}$ is partial-trace believing.*

Proof: From [Eng85] we know that most notions of process equivalence coincide for partial trace determinate processes. In particular, two processes are partial trace equivalent iff they are bisimilar. From [Blo89], we know that if P and Q are bisimilar, then they are partial trace congruent in all GSOS languages; in particular, they are partial trace congruent in $\mathcal{L}$ as desired. $\triangleup$

It thus behooves us to ask how powerful a language can be and still force all processes to be partial-trace determinate. We show below that this is undecidable, and so a full answer is impossible. We give a useful partial answer, by identifying a class of languages which produce only *term* determinate processes; notice that term determinacy implies partial-trace determinacy (and, indeed, implies all forms of determinacy under Definition 2.6).

Briefly, we shall enforce determinacy in two ways. In many circumstances, we can insist that only one rule should apply to any given term; that is, if there are two rules giving $\alpha(\vec{P})$ a -moves, only one of them can fire. This requirement forbids many useful operations, so we allow two rules to apply – but insist that the results of the two rules are the same. These requirements make all terms in the language term-determinate.

Definition 3.3 *Let Ξ and Ξ' be sets of transition formulas of the form $X \xrightarrow{a} Y$ and $X \xrightarrow{b}$; furthermore, suppose that all the source variables in Ξ and Ξ' are disjoint from the target variables. Ξ and Ξ' are non-overlapping iff there is a formula of the form $X \xrightarrow{a} Y$ in one of them and $X \xrightarrow{a}$ in the other.*

So, the sets $\{X \xrightarrow{a} Y\}, \{X \xrightarrow{a}\}$ are non-overlapping, but $\{X \xrightarrow{a} X'\}, \{Y \xrightarrow{a} Y'\}$ overlap. (In particular, the CCS parallel rules overlap.) Note that, if Ξ and Ξ' are non-overlapping, then there is no assignment of terms to variables which makes both Ξ and Ξ' true simultaneously.

If all pairs of rules which give each operator α the possibility of an a -behavior are non-overlapping, then the language is term determinate — at most one rule can be applied to any term. However, this is not a complete description of term-determinate languages. For example, consider α defined by the rules of Figure 2. The two rules overlap, and indeed there are two different kinds of a -transitions possible for $\alpha(b + cP')$. However, $\alpha(b + cP')$ is term determinate if P' is, as both transitions end up at P' .

Definition 3.4 *Two rules ρ^0, ρ^1 giving α a -transitions act thoroughly identical iff, whenever $\vec{P}$ is a vector of term determinate terms, and each rule*

$$\frac{X \xrightarrow{b} Y, \quad X \xrightarrow{c} Y'}{\alpha(X) \xrightarrow{a} Y'} \quad \frac{X \xrightarrow{c} Y, \quad X \xrightarrow{c} Y', \quad X \xrightarrow{a} \quad X \xrightarrow{d}}{\alpha(X) \xrightarrow{a} Y}$$

Figure 2: Overlapping yet Term Determinate

ρ^i gives the transition $\alpha(\vec{P}) \xrightarrow{a} Q^i$, then $Q^0 = Q^1$.

Figure 2 illustrates all of the complexities of acting thoroughly identical. Whenever both rules can fire on a single term, the results must be the same. This means that the variables Y_{ij} that actually appear in the targets $C[\vec{X}, \vec{Y}]$ must come from the same kind of transition (α would fail to act thoroughly identical if both targets were Y , as Y in the left rule is a b -child of X and in the right rule a c -child; thus $\alpha(bb + cc)$ would not be term determinate). The following lemma gives a syntactical test.

By term determinacy of the arguments, it suffices to consider rules in which there is at most one antecedent $X_i \xrightarrow{a} Y_{ia}$ for each i and a . That is, the $X \xrightarrow{c} Y'$ in the right-hand rule of Figure 2 would be irrelevant even if it did appear in the target; when X is term determinate, Y' must always be instantiated with the same term as Y in that rule.

Bearing this in mind, we can be somewhat more specific about the format of GSOS rules which are intended only for term determinate processes. Rather than having antecedents of the form $X_i \xrightarrow{a_{ij}} Y_{ij}$, we can have antecedents of the form $X_i \xrightarrow{a_{ij}} Y_{ia_{ij}}$, where the a_{ij} 's are distinct (for i fixed). That is, there is at most one a -child of X_i asked for in a given rule, and each child variable Y_{ia} tells which action a lead to it as well as which parent variable X_i bore it. A GSOS language of this form is called *pedigreed*. This makes the following lemma trivial:

Lemma 3.5 *Let*

$$\frac{X_i \xrightarrow{a_{ij}} Y_{ia_{ij}}^{(w)}, \quad X_i \xrightarrow{b_{ik}^{(w)}}}{\alpha(\vec{X}) \xrightarrow{c} C^{(w)}[\vec{X}, \vec{Y}]}$$

for $w = 0, 1$ be two *pedigreed, overlapping, satisfiable rules giving c-behavior to α* . Then the rules act thoroughly identical if the terms $C^{(0)}[\vec{X}, \vec{Y}]$ and $C^{(1)}[\vec{X}, \vec{Y}]$ are syntactically identical.

The converse of this lemma is true if the language is powerful enough to express a sufficiently complicated process which can trigger both rules.

Such processes are trivial to build in CCS, with unrestricted use of nondeterministic choice; however, nondeterministic choice must be restricted in a determinate language. The timid choice operation (3.3.1) suffices to build them.

Definition 3.6 *A GSOS language is nicely determinate iff, whenever ρ and ρ' are distinct rules with the same operation and action and have overlapping antecedents, then they act thoroughly identical.*

Definition 3.3 and Lemma 3.5 give a procedure for testing pedigreed GSOS languages for nice determinacy. Typical examples of non-nicely-determinate operators are the CCS choice and parallelism rules.

We say that P is *one-step* $\equiv$ *determinate* iff, whenever $P \xrightarrow{a} P_1$, $P \xrightarrow{a} P_2$, then $P_1 \equiv P_2$. Then it is easy to prove by induction that:

Lemma 3.7 *Suppose that all terms in a language are one-step $\equiv$ determinate. Then all terms are $\equiv$ determinate.*

Lemma 3.8 *If P is a process in a nicely determinate language, then P is term determinate.*

Proof: We proceed by induction on terms, using Lemma 3.7 to reduce the problem to one-step term determinacy. The base case is when P is a constant; that is, an operator which takes no arguments. Without arguments, there can be no antecedents. The null set overlaps itself, and so any two rules giving a -behavior to P act thoroughly identical. In particular, there is at most one a -descendant of P for any a .

For the induction step, assume that $P = \alpha(\vec{P})$, and that each P_i is one-step term determinate. Suppose that $P \xrightarrow{a} P^1$, $P \xrightarrow{a} P^2$, where the transition to P^i is enabled by rule ρ^i . If $\rho^1 = \rho^2$, then P^1 and P^2 must be identical because of the one-step term determinacy of $\vec{P}$. If $\rho^1 \neq \rho^2$, then the two rules must overlap. So, by hypothesis, they act thoroughly identical, and therefore $P^1 = P^2$. $\triangleup$

3.3 Discussion

3.3.1 Using Determinate Operations

It is possible to define determinate variants of most standard operations. For example, we may define the *timid choice operator* (related to the *angelic*

choice operator of [MO]). A full determinate variant of CCS will appear elsewhere.

$$\frac{X_1 \xrightarrow{a} Y_1, \quad X_2 \xrightarrow{a}}{X_1 \oplus X_2 \xrightarrow{a} Y_1} \quad \frac{X_1 \xrightarrow{a}, \quad X_2 \xrightarrow{a} Y_2}{X_1 \oplus X_2 \xrightarrow{a} Y_2} \quad \frac{X_1 \xrightarrow{a} Y_1, \quad X_2 \xrightarrow{a} Y_2}{X_1 \oplus X_2 \xrightarrow{a} Y_1 \oplus Y_2}$$

3.3.2 On True Sequencing

True sequencing is a theoretically appealing operation, with a simple informal definition. However, implementing it can be rather tricky – it may involve deadlock detection or worse to tell when P is stopped, as P may not be able to tell itself. So, most core languages for concurrency do not use true sequencing in its full power. CCS makes do with prefixing – true sequencing in the special case where the first process is a simple process guaranteed to terminate. CSP has a sequencing operation which approximates true sequencing in most cases, but still can be implemented easily.

CSP, among other languages, distinguishes between *successful* and *unsuccessful* termination. Programs which *successfully* finish announce that fact by sending a distinguished $\checkmark$ action; those that terminate unsuccessfully simply stop. The sequencing operator catches $\checkmark$ actions and treats them as signals that processes are finished. The rules for CSP/ACP sequencing are:

$$\frac{X \xrightarrow{a} Y, \quad a \neq \checkmark}{X \cdot, Z \xrightarrow{a} Y \cdot, Z} \quad \frac{X \xrightarrow{\checkmark} Y, \quad Z \xrightarrow{a} W}{X \cdot, Z \xrightarrow{a} W}$$

If $\cdot$, is to approximate true sequencing as closely as possible, rules involving $\checkmark$ must be carefully crafted so that processes produce $\checkmark$ only when they are finished. The $+$ rule for $\checkmark$ is different from the other rules for $+$ given in Figure 1:

$$\frac{X_1 \xrightarrow{\checkmark} Y_1, \quad X_2 \xrightarrow{\checkmark} Y_2}{X_1 + X_2 \xrightarrow{\checkmark} \mathbf{0}}$$

This differs from true sequencing in that, if P is deadlocked (e.g., $a|_{\{a,b\}}b$), then $P;Q$ will run Q but $P \cdot, Q$ will also deadlock. This form of sequencing is almost as useful as true sequencing, and (as it does not require deadlock detection) easier to implement. However, it infringes on the purity of intent of the partial trace model: the actions in the partial traces were precisely the actions that the processes took. The $\checkmark$ action is more of an encoding of an aspect of process state, which one may not wish to directly consider.

3.4 Partial Trace Determinacy is Undecidable

We presented a simple syntactic discipline which guarantees that a language is partial trace determinate. It is clear that this scheme is incomplete; for example, the following operations (both of which are effectively identity functions) may be added to any partial trace determinate language and the result will still be partial trace determinate.

$$\frac{X \xrightarrow{a} Y}{\begin{array}{l} i(X) \xrightarrow{a} i(Y) \\ j(X) \xrightarrow{a} i(Y) \\ j(X) \xrightarrow{a} Y \end{array}} \quad (3)$$

The process $i(P)$ has exactly the same synchronization tree as P . The process $j(P)$ has the synchronization tree of $P + P$, and thus has the same partial traces P . However, it is not nicely determinate.

Unsurprisingly, it is undecidable in general whether or not a language is partial-trace deterministic. Let $T(n)$ be a deterministic process which simulates the n 'th Turing machine in some convenient enumeration on blank tape, producing b_0 actions while it is running and stopping when and if the Turing machine halts.⁵ Consider the family of languages $\mathcal{L}_n$, which include the following operations. (a ranges over all actions; b_0 is constant.) The forking operator j is defined by the rules

$$j(X) \xrightarrow{b_0} X \quad j(X) \xrightarrow{b_0} \beta(X, T(n)).$$

$\beta(X, Z)$ is an eccentric operator which behaves like X as long as Z emits b_0 's. After Z is finished, $\beta(X, Z)$ emits a b_0 and stops.

$$\frac{X \xrightarrow{a} Y, \quad Z \xrightarrow{b_0} Z'}{\beta(X, Z) \xrightarrow{a} \beta(Y, Z')} \quad \frac{Z \xrightarrow{b_0} \mathbf{0}}{\beta(X, Z) \xrightarrow{b_0} \mathbf{0}}$$

If the n 'th Turing machine diverges on blank tape, then $\beta(P, T(n))$ simply simulates P , and so $j(P)$ is identical to $b_0.P$; that is, $\mathcal{L}_n$ is deterministic.

However, if the n 'th Turing machine halts with $T(n)$ producing k b_0 's, then $j(a^{k+1})$ can behave either like $b_0 a^{k+1}$ or like $b_0 a^k b_0$, and thus $\mathcal{L}_n$ is not deterministic.

⁵Programming such a Turing machine is an easy exercise in most process calculi. It is not necessary to produce only one b_0 for each step of Turing machine computation.

3.5 Silent Moves vs. Partial Trace Congruence

Silent moves, taken naïvely, destroy the previous theory. We proceed in the simplest way possible: as we are only concerned with visible actions, we remove all τ 's from the traces. If s is a string of actions and τ 's, we define $s \downarrow \tau$ to be the string s with all the τ 's removed. We define the *de- τ ed partial trace set of P* , $\text{ptr}_\tau(P)$, to be $\{s \downarrow \tau \mid s \in \text{ptr}(P)\}$.

Let Ω be a process which diverges by doing τ -moves; e.g., $\Omega = \tau ; \Omega$. We have $\text{ptr}_\tau(\mathbf{0}) = \text{ptr}_\tau(\Omega) = \{\langle \rangle\}$. However, $\mathbf{0}$ and Ω behave quite differently under sequencing: $\text{ptr}_\tau(\mathbf{0} ; a) = \{\langle \rangle, a\}$, yet $\text{ptr}_\tau(\Omega ; a) = \{\langle \rangle\}$.

It is reasonably straightforward to give a semantics which accounts for this phenomenon; *e.g.*, add some divergence information to partial traces. In any event, our basic goal of having partial traces be an adequate model has been violated.

4 Positive Rules

4.1 Straight Tyft Languages

In [GV89], Groote and Vaandrager define *tyft languages*, a quite powerful form of SOS language which enjoy sanity properties only slightly weaker than those of GSOS languages. (Tyft languages may be countably branching, and the transition relation may be non-recursive). Rules in tyft languages have the form

$$\frac{\{t_i \xrightarrow{a_i} Y_i\}}{f(\vec{X}) \xrightarrow{b} t} \quad (4)$$

where the t_i 's and t are terms with free variables in the set $\{\vec{X}, \vec{Y}\}$, and the variables in $\vec{X}$ and $\vec{Y}$ are all distinct. All of the positive rules in this paper are tyft rules; indeed, rules in the more restrictive *xyft* format in which the t_i 's in the antecedent are X_i 's and Y_i 's. An interesting tyft operation which is not definable by GSOS rules is the skip-an- a operation:

$$\frac{X \xrightarrow{a} Y, \quad Y \xrightarrow{b} Z}{X/a \xrightarrow{b} Z} \quad (5)$$

It is in general inconsistent to add negative rules to tyft languages; see [Blo89].

In a tyft language $\mathcal{G}$, $P \xrightarrow{a} Q$ iff there is a proof of the following sort for this fact. A proof for $P \xrightarrow{a} Q$ is a well-founded tree, with nodes labelled by transitions $R \xrightarrow{a} S$, such that:

1. The root is labelled $P \xrightarrow{a} Q$; and
2. Let q be an arbitrary node and $R \xrightarrow{a} S$ its label. Then there is a rule $\rho = \chi_1, \dots, \chi_n \text{ /- } \chi$ and a substitution (map from variables to terms) σ , such that:
 - (a) $\sigma(\chi) = R \xrightarrow{a} S$
 - (b) The children q_i of q are in 1-1 correspondence with the antecedents χ_i of ρ , and $\sigma(\chi_i)$ is the label of q_i .

Tyft languages in full generality are allowed to make copies of processes, and thus are able to discover a good deal about their branching behavior. If partial traces are to be adequate, we cannot allow this.

Definition 4.1 *A tyft rule*

$$\frac{\{t_i \xrightarrow{a_i} Y_i\}}{f(\vec{X}) \xrightarrow{b} t}$$

is straight iff it is finite in both rules and antecedents, every variable in t is either an X_i or a Y_i , and no variable occurs more than once in $\{t, t_1, \dots, t_n\}$. A tyft language is straight iff all rules are straight.

For example, the $\cdot/a$ operation above is straight; $X \xrightarrow{a} Y, X \xrightarrow{b} Z \text{ /- } \delta(X) \xrightarrow{a} Z$ is not straight (as it looks at two behaviors of X), and neither is

$$X \xrightarrow{a} Y, \quad Y \xrightarrow{b} Z \tag{6}$$

(as it uses Y twice).

Theorem 4.2 *Every language defined by straight tyft rules is partial-trace believing.*

Proof: A straightforward consequence of the theory which we shall develop in Section 4.2. In particular, we prove Lemma 4.6, which shows that each partial trace of $C[P]$ is the consequence of some partial trace of P . Hence, if P and Q have the same partial traces, they must be congruent. $\triangle$

4.2 Theory of Straight Tyft Languages

We wish to show that, if P and Q have the same partial traces, then they have the same partial traces in all contexts. First we reduce the problem to the case of some fairly simple contexts (those in which the hole occurs at most once); then we prove it for those contexts by a marking argument.

Definition 4.3 *A context $C[X]$ is solitary if X occurs at most once in the term $C[X]$.*

The following fact holds in considerable generality.

Lemma 4.4 *Let $\sim$ be an equivalence relation between processes. Suppose that P and Q are $\sim$ -congruent in all solitary contexts. Then P and Q are $\sim$ -congruent in all contexts.*

Proof: This proof is similar to Milner's proof of the Context Lemma. We illustrate the method by $C[X] = X|X|X$. Lines connected by “=” are syntactic equalities; lines connected by $\sim$ follow from the solitary congruence of P and Q from the contexts $C_i[X]$.

$$\begin{aligned}
C[P] &= P|P|P \\
= C_1[P] &= P|P|P & C_1[X] &=_{rdef} X|P|P \\
\sim C_1[Q] &= Q|P|P \\
= C_2[P] &= Q|P|P & C_2[X] &=_{rdef} Q|X|P \\
\sim C_2[Q] &= Q|Q|P \\
= C_3[P] &= Q|Q|P & C_3[X] &=_{rdef} Q|Q|X \\
\sim C_3[Q] &= Q|Q|Q \\
= C[Q] &= Q|Q|Q
\end{aligned} \tag{7}$$

$\triangle$
+

Now, we investigate solitary contexts. We start with a context $C[X]$ in the language $\mathcal{L}$. We want to watch $C[P]$ as it evolves, and we'll need to keep

track of P and its children. As P and its children may occur as subterms of $C[X]$ — as in the previous lemma — we will have to mark the one we want to observe.

Borrowing the notion of “labelled reduction” [Bar81, Ch.14], we define a family of *marker identities*, operators $\#_n(P)$, which will let us keep track of the behavior of processes. The marker identities are given behavior by the rules $X \xrightarrow{a} Y \text{ /- } \#_n(X) \xrightarrow{a} \#_{n+1}(Y)$ for all a and n . It is trivial to show that $\#_n(\cdot)$ is an identity operator on synchronization trees.

Lemma 4.5 *Suppose $C[Z]$ is a solitary context and P a process over $\mathcal{L}$. Let π be a proof (in $\mathcal{L} + \#(\cdot)$) that $C[\#_m(P)] \xrightarrow{a} Q$. Then:*

- #1:** *If $C[Z]$ contains no Z , then there are no subterms of the form $\#(\cdot)$ in any term in π .*
- #2:** *Let R be any term appearing (as source or target of any node) in π . Then there is at most one subterm R of the form $\#(\cdot)$.*
- #3:** *Fix n . There is at most one node of π of the form $\#_n(V_n) \xrightarrow{b} \#_{n'}(V_{n'})$. Furthermore, if there is such a node, then $n' = n + 1$, and there are such nodes for $\#_m(\cdot)$ through $\#_n(\cdot)$.*

Proof: By induction on proofs. Suppose that it holds for all proofs smaller than π ; we show that it holds for π . The label at the base of π is $C[\#_m(P)] \xrightarrow{a} Q$. Suppose that the rule for this is ρ , which is

$$\frac{t_1 \xrightarrow{a_1} Y_1, \dots, t_r \xrightarrow{a_r} Y_r}{\alpha(\vec{X}) \xrightarrow{a} t}$$

Note that either $C[Z] = Z$ and ρ is the $\#_m(\cdot)$ rule, or ρ is a rule of $\mathcal{L}$.

If $C[Z] = Z$ and ρ is the $\#_m(\cdot)$ rule, then $Q = \#_{m+1}(Q')$ where $P \xrightarrow{a} Q'$. As there are no $\#(\cdot)$'s in P , there are none in this proof save the leading one in Q , and the desired properties follow trivially.

Otherwise, ρ is a rule of $\mathcal{L}$, and $C[Z] = \alpha(C_1[Z], \dots, C_c[Z])$. From the definition of “proof”, there is a substitution σ such that: $\sigma(X_j) = C_j[\#_m(P)]$ and $\sigma(t) = Q$, and each of the subproofs of π is a proof π_i of $\sigma(t_i) \xrightarrow{a_i} \sigma(Y_i)$.

We show the three desired properties in order. #1 is trivial. For #2 and #3, we note that, as $C[Z]$ is solitary, Z occurs in at most one of the $C_j[Z]$'s. If it does not occur in any, then #1 gives us #2 and #3 immediately; so

we assume that it does, say in $C_{j_0}[Z]$. Of course, it occurs exactly once in $C_{j_0}[Z]$. If there are no hypotheses in which $X_{j_0} \subseteq t_i$, then once again #2 and #3 are trivial; so we assume that there is one in t_{i_0} . (There cannot be more than one by straightness.) By induction (and the fact that they hold for $C[\#_m(P)]$) we know #2 and #3 for all terms and nodes in π except for Q .

Say that a variable $v = X_j$ or Y_i is an *immediate precursor* of Y_k if there is a hypothesis of ρ of the form $t_k \xrightarrow{a_k} Y_k$ with $v \subseteq t_k$. Also, say that v is a *precursor* of Y_k if it is an immediate precursor of Y_k or any of Y_k 's precursors. "Postcursor" is the converse of "precursor." Note that the straightness requirement implies that any variable which is a precursor of any other variable cannot occur in t . Note also that $\{Y_k | X_j \text{ is a precursor of } Y_k\}$ is totally ordered by the precursor relation. Finally, note that if a subterm of $\sigma(Y_k)$ is marked, then X_{j_0} must be a precursor of Y_k .

Suppose that there are two marked subterms of Q . As there are no markers in t , these terms must have appeared by substitution for some Y_{i_1} and Y_{i_2} . Now, X_{j_0} must be a precursor of or equal to both of these, and so one must be a precursor of the other. But by straightness, this is impossible. So there can be at most one marked subterm of Q ; we have shown #2.

Finally, #3 follows easily by induction, following the maximal chain of postcursors starting from X_{j_0} . $\triangle$

Finally, we come to the theorem

Lemma 4.6 *Suppose that, in some straight tyft language, $C[P] \xrightarrow{a} Q$. Then there is a string s , process P' , and context C' such that $P \xrightarrow{s} P'$, $Q = C'[P']$, and for any R , if $R \xrightarrow{s} R'$, then $C[R] \xrightarrow{a} C'[R']$.*

Proof: Suppose that π is a proof that $C[P] \xrightarrow{a} Q$. As $\#_0(P) \xleftrightarrow{\epsilon} P$, we know by the main result of [GV89] that $C[\#_0(P)] \xrightarrow{a} Q'$. By a somewhat deeper analysis, we see that we can construct a proof π' which looks like π , with the reducts of P marked, and with a stratum of $\#.(.)$ rules in some branches. Let π' be this proof; note that Lemma 4.5 applies to π' .

The nodes that #3 gives are labelled with transitions which may be hooked together:

$$\#_0(P) \xrightarrow{a_1} \#_1(P_1) \xrightarrow{a_2} \dots \xrightarrow{a_n} \#_n(P_n) \quad (8)$$

Let $s = a_1 a_2 \dots a_n$ and $P' = P_n$. Let $C'[Z]$ be Q' with the marked term (if it exists) replaced by Z . It is clear that $P \xrightarrow{s} P'$ and $Q = C'[P']$.

We must now show that, if $R \xrightarrow{s} R'$, then $C[R] \xrightarrow{a} C'[R']$.

Let π'' be π' truncated above all nodes of the form $\#_n(V) \xrightarrow{a} \#_{n+1}(V')$ (so that those nodes are some of the leaves of π''). Let $R_0 = R$, and $R_i \xrightarrow{a_i} R_{i+1}$ be the terms along $R \xrightarrow{s} R'$. We construct a proof π''' by replacing $\#_k(P_k)$ in π'' by R_k , and above each leaf which in π'' was labelled $\#_k(P_k) \xrightarrow{a_k} \#_{k+1}(P_{k+1})$, attach the proof that $R_k \xrightarrow{a_k} R_{k+1}$. It is straightforward to check that this is indeed a proof that $C[R] \xrightarrow{a} C'[R']$. $\triangle$

4.3 Silent Moves

The theory of SOS semantics for languages with silent moves is not yet established. We propose two definitions; but it is beyond the scope of this paper to sort out the distinctions. In all cases, reasonably simple definitions of “straight,” along the lines Definition 4.1, give languages which respect partial traces.

A straightforward way to design SOS rules which respect the silence of silent moves is to require *patience*. Intuitively, a language is patient if, whenever a term is concerned about the behavior of an argument, it will allow that argument to execute an arbitrary number of τ -moves [Blo90, Vaa91]. The formal definition of patience in tyft languages is simple but drastic – that is, a patient tyft language is not a tyft language. For GSOS languages, we may define “patient GSOS languages” which are still GSOS languages.

Recall the standard definition: if $a \neq \tau$, then $P \xrightarrow{a} Q$ if $P \xrightarrow{\tau} \dots \xrightarrow{\tau} P' \xrightarrow{a} Q' \xrightarrow{\tau} \dots \xrightarrow{\tau} Q$.

Definition 4.7 *A patient τ^* tyft language with actions in Act is syntactically the same as a tyft language with actions in $\text{Act} \cup \{\tau\}$, subject to the constraint that τ never appear as an action in an antecedent. The operational semantics are different; we interpret $\xrightarrow{a}$ as $\xrightarrow{a}$ in the antecedents of rules.*

In this way, we allow subprocesses to take any finite number of τ -moves whenever they wish. For example, the $\cdot/a$ operation in (5) is allowed to skip τ 's after the a : we have $(a\tau\tau\tau bP)/a \xrightarrow{b} P$.

Proofs get somewhat more complex, as they may include long proofs of chains of τ -moves. However, the theory of Section 4.1 still holds, mutatis mutandis.

Theorem 4.8 *De- τ ed partial traces are an adequate semantics for all straight silent-move tyft languages.*

Proof: Define markers $\#(\cdot)$ as before (and interpret it non-patiently). Lemma 4.5 still holds, with roughly the same proof. Lemma 4.6 follows as well, with somewhat more fiddling. The full theorem follows straightforwardly. $\triangle$

However, this change is fairly drastic. Silent-move tyft languages are not tyft languages; the operational semantics is different, and there is no guarantee that a theorem about one will also be a theorem about the other. We propose a subclass of tyft languages which behave the same as silent-move tyft languages.

Consider the operator τ^{-1} which runs its argument for a few τ 's, an a , and some more τ 's:

$$\frac{X \xrightarrow{a} Y}{\tau^{-1}(X) \xrightarrow{a} Y} \quad \frac{X \xrightarrow{\tau} X', \quad \tau^{-1}(X') \xrightarrow{a} Y}{\tau^{-1}(X) \xrightarrow{a} Y} \quad \frac{\tau^{-1}(X) \xrightarrow{a} Y', \quad Y' \xrightarrow{\tau} Y}{\tau^{-1}(X) \xrightarrow{a} Y}$$

Definition 4.9 *A patient τ^{-1} tyft language is a tyft language including the τ^{-1} operator, and in which all antecedents of other rules have the form $\tau^{-1}(t_i) \xrightarrow{a_i} y_i$ where $a_i \neq \tau$.*

Patient τ^{-1} and silent-move tyft languages have the same expressive power at a very fine level; it is straightforward to define a translation between languages and between proof trees so that, whenever $P \xrightarrow{a} Q$ in one form of tyft language, then $P \xrightarrow{a} Q$ in the other with essentially the same proof. The following theorem is a simple corollary of that translation:

Theorem 4.10 *De- τ 'ed partial trace equivalence is adequate for straight patient tyft languages.*

Patient tyft languages, either with $\xrightarrow{a}$ interpreted as $\xRightarrow{a}$ or with τ^{-1} in antecedents, are distressingly powerful. For example, $(a\tau\tau\tau b)/a \xrightarrow{b} \mathbf{0}$ despite the fact that the process $(a\tau\tau\tau b)$ is a rather slow one. The ability of tyft languages to collapse arbitrary numbers of τ s into nothing seems technically acceptable but philosophically disturbing.

In many circumstances, we do not need the full power of tyft languages. In particular, having arbitrary terms as antecedents may be rather expensive; it is often simpler to simply have antecedents of the form $X \xrightarrow{a} Y$. [Vaa91] gives a theorem concerning a subclass of straight GSOS languages for which partial trace equivalence is adequate.

5 Conclusion

We have presented two independent cases in which partial traces were fully abstract. The first case admitted some fairly powerful operations, including true sequencing and broadcasting, and partial trace equivalence captured total correctness; however, it required processes to be determinate and it did not extend to silent actions. The second case does not require determinacy, and does extend to silent actions; however, it does not allow sequencing, and does not capture total correctness. It seems likely that this is the best that can be achieved; partial traces are so weak a semantics that, without drastic restrictions on processes, they cannot give total correctness.

5.1 Open Problems

A number of questions remain to be investigated. The most perplexing of these is that the results of the first half of this study all hold for process *equivalence*, but not process *approximation* – where we take the standard view that P is a partial-trace approximation of Q if every partial trace of P is also a partial trace of Q . That is, we know that if $\text{ptr}(P) = \text{ptr}(Q)$, then $\text{ptr}(C[P]) = \text{ptr}(C[Q])$; but for determinate GSOS languages, it need not be the case that $\text{ptr}(P) \subseteq \text{ptr}(Q)$ implies $\text{ptr}(C[P]) \subseteq \text{ptr}(C[Q])$. For example, $\text{ptr}(\mathbf{0}) \subseteq \text{ptr}(a)$, but $\text{ptr}(\mathbf{0}; b) \not\subseteq \text{ptr}(a; b)$. There surely is a partial order on trace sets for determinate processes which describes process approximation — the full synchronization tree can be recovered from the trace sets — but it remains to be adequately described. (The results in the second half of the study do hold for partial trace approximation as well.)

Partial trace equivalence is the simplest usable notion of process equivalence, but there are several others which are not much harder. Later studies will cover at least completed traces and CSP-style traces with end-markers, and probably other models as well. We conjecture that the results in this paper continue to hold in a straightforward way in the setting of infinitary traces.

6 Acknowledgments

I gratefully acknowledge many comments and much assistance from Frits Vaandrager, Georges Lauri, and Marta Kwiatkowska. I also thank my partner Vicki Borah for the owl and other encouragement.

References

- [AV90] Samson Abramsky and Steven Vickers. Quantales, observational logic, and process semantics. Research Report DOC 90/1, Imperial College, London, 1990.
- [Bar81] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic*. North-Holland, 1981. Revised Edition, 1984.
- [BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.
- [BC89] G. Boudol and I. Castellani. Permutation of transitions: an event structure semantics for CCS and SCCS. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *REX School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Noordwijkerhout, volume 354 of *Lect. Notes in Computer Sci.*, pages 411–427. Springer-Verlag, 1989.
- [BG90] R.N. Bol and J.F. Groote. The meaning of negative premises in transition system specifications. Report CS-R9054, CWI, Amsterdam, 1990.
- [BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.
- [BIM88] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can’t be traced (preliminary report). In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 229–239, 1988. Also appears as MIT Technical Memo MIT/LCS/TM-345.
- [BIM90] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can’t be traced. Technical Report TR 90-1150, Cornell, August 1990.
- [BK84] J. A. Bergstra and J. W. Klop. Algebra of communicating processes. Technical Report CS-R8421, Centre for Mathematics and Computer Science, Amsterdam, 1984.

- [BK86] J.A. Bergstra and J.W. Klop. Algebra of communicating processes. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Mathematics and Computer Science*, CWI Monograph 1, pages 89–138. North-Holland, Amsterdam, 1986.
- [Blo89] Bard Bloom. *Ready Simulation, Bisimulation, and the Semantics of CCS-Like Languages*. PhD thesis, Massachusetts Institute of Technology, August 1989.
- [Blo90] Bard Bloom. Strong process equivalence in the presence of hidden moves (preliminary report). (Unpublished), July 1990.
- [BM90] Bard Bloom and Albert R. Meyer. Experimenting with process equivalence. In Dr. Marta Kwiatkowska, editor, *Proceedings of the International BCS-FACS Workshop on Semantics for Concurrency*, pages 189–201, Leicester, U.K., July 1990.
- [dNH84] R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Sci.*, 34(2/3):83–133, 1984.
- [dS85] Robert de Simone. Higher-level synchronising devices in MEIJE-SCCS. *Theoretical Computer Sci.*, 37(3):245–267, 1985.
- [Eng85] J. Engelfriet. Determinacy $\rightarrow$ (observation equivalence = trace equivalence). *Theoretical Computer Sci.*, 36(1):21–25, 1985.
- [GG89] R.J. Glabbeek and U. Goltz. Equivalence notions for concurrent systems and refinement of actions. Arbeitspapiere der GMD 366, Gesellschaft für Mathematik und Datenverarbeitung, Sankt Augustin, 1989. An extended abstract appeared in: Proceedings 14th Symposium on Mathematical Foundations of Computer Science, Porąbka-Kozubnik, Poland, August/September 1989 (A. Kreczmar and G. Mirkowska), LNCS 379, Springer-Verlag, pp. 237–248.
- [Gla90] R.J. van Glabbeek. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90*, Amsterdam, volume 458 of *Lect. Notes in Computer Sci.*, pages 278–297. Springer-Verlag, 1990.
- [GV89] Jan Friso Groote and Frits Vaandrager. Structured operational semantics and bisimulation as a congruence (extended abstract). In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca,

editors, *Automata, Languages and Programming: 16th International Colloquium*, volume 372 of *Lect. Notes in Computer Sci.* Springer-Verlag, 1989.

- [Hen83] Matthew Hennessy. Synchronous and asynchronous experiments on processes. *Information and Computation*, 59(1–3):36–83, 1983.
- [Hoa78] C. A. R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, 1978.
- [KLP90] Stefano Kasangian, Anna Labella, and Alberto Petrorossi. Observers, experiments, and agents: A comprehensive approach to parallelism. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes*, pages 375–406, 1990. LNCS 469.
- [Maz89] Antoni Mazurkiewicz. Basic notions of trace theory. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lect. Notes in Computer Sci.* Springer-Verlag, 1989.
- [Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Sci.*, 25(3):267–310, 1983.
- [Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, New York, 1989.
- [MO] Michael Mislove and Frank Oles. A simple language supporting angelic nondeterminism and parallel composition. (Presented at MFPS 91; to appear in the proceedings).
- [Plo81] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, Denmark, 1981.
- [PP90] G.D. Plotkin and V.R. Pratt. The resolving power of multiple observers (extended abstract). (The final version has not yet appeared, to my knowledge), 1990.
- [Udd86] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, 1:197–204, 1986.

- [Vaa88] Frits Vaandrager. Determinism $\rightarrow$ (event structure isomorphism = step sequence equivalence). Technical Report CS-R8839, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, October 1988.
- [Vaa91] Frits Vaandrager. On the relationship between process algebra and input/output automata. (Extended abstract for LICS 91), April 1991.
- [vGW89] Rob van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. In G. X. Ritter, editor, *Information Processing 89: Proceedings of the IFIP 11th World Computer Congress*, pages 613–618. North-Holland, August 1989.
- [Wal90] David Walker. Bisimulation and divergence. *Information and Computation*, 85(2):202–241, April 1990.