Bisimulation Can't Be Traced

BARD BLOOM

Cornell University, Ithaca, New York

SORIN ISTRAIL

Sandia National Laboratories, Albuquerque, New Mexico

AND

ALBERT R. MEYER

Massachusetts Institute of Technology, Cambridge, Massachusetts

Abstract. In the concurrent language CCS, two programs are considered the same if they are *bisimilar*. Several years and many researchers have demonstrated that the theory of bisimulation is mathematically appealing and useful in practice. However, bisimulation makes too many distinctions between programs. We consider the problem of adding operations to CCS to make bisimulation fully abstract. We define the class of GSOS operations, generalizing the style and technical advantages of CCS operations. We characterize GSOS congruence in as a bisimulation-like relation called *ready simulation*. Bisimulation is strictly finer than ready simulation, and hence not a congruence for any GSOS language.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*algebraic approaches to semantics*; *operational semantics*; I.6.2 [**Simulation and Modeling**]: Simulation Languages

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Bisimulation, structural operational semantics, process algebra, CCS

The work of B. Bloom was supported by a National Science Foundation (NSF) Fellowship, also NSF Grant No. 85-11190-DCR and Office of Naval Research (ONR) grant no. N00014-83-K-0125.

The work of S. Istrail was supported by NSF Grant CCR 88-1174. This work performed at Sandia National Laboratories supported by the U.S. Department of Energy under contract DE-AC04-76DP00789.

The work of A. R. Meyer was supported by NSF Grant No. 85-1190-DCR and by ONR grant no. N00014-83-K-0125.

Authors' addresses: B. Bloom, Cornell University, Department of Computer Science, Upson Hall, Ithaca, NY 14853; email: bard@cs.cornell.edu; S. Istrail, Sandia National Laboratories, email: scistra@cs.sandia.gov; and A. R. Meyer, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA 02139, email: meyer@theory.lc.mit.edu

1. *Introduction*

One of the most basic things that a programming-language semantics should give is a notion of *program equivalence*: a statement telling when two programs do the same thing. Frequently, there are many choices of a notion of program equivalence, and it is not clear how to choose among them. We explore some criteria for selecting one notion over another.

Two concurrent programming languages, Milner's CCS [Milner 1980, 1983, 1989] and Hoare's CSP [Hoare 1978, 1985], share the premise that the meaning of a process is fully determined by a *synchronization tree*, namely, a rooted, unordered tree whose edges are labeled with symbols denoting basic *actions* or events. These trees are typically specified by a Structured Operational Semantics (SOS) in the style of Plotkin [1981] or by some other effective description, and so are in fact recursively enumerable trees. Both theories further agree that synchronization trees are an *over*specification of process behavior, and certain distinct trees must be regarded as equivalent processes. The notable difference in the theories is that the CCS notion of bisimulation yields finer distinctions among synchronization trees; that is, CSP identifies trees that CCS considers different, but not conversely.

In CSP, process distinctions can be understood as based on observing *completed traces*, namely, *maximal* sequences of visible actions performed by a process. Two trees are *trace equivalent* iff they have the same set of traces. Given any set of operations on trees, *trace congruence* is defined to be the coarsest congruence with respect to the operations which refines trace equivalence. Thus, two CSP processes P and Q are distinguished iff there is some CSP context $C[X]$ and string s such that only one of $C[P]$ and $C[Q]$ has s as a trace. This explanation of when two synchronization trees are to be identified is thoroughly elaborated in de Nicola and Hennessy's [1984] *test equivalence* system. On the other hand, two CCS processes are distinguished according to an "interactive" game-like protocol called *bisimulation*. Indistinguishable CCS processes are said to be *bisimilar*.

A standard example is the pair of trees in Figure 1, $a(b + c)$ and $(ab + ac)$, which are trace equivalent, but not CSP trace congruence, viz., in both CSP and CCS they are distinct processes. Similarly, the trees of Figure 2,

$$(abc + abd) \qquad \text{and} \qquad a(bc + bd) \tag{1}$$

are CSP trace congruent but not bisimilar, viz., equal in CSP but considered distinct in CCS [Brookes 1983; Pneuli 1985]. The tree-based approach is developed in Brookes et al. [1984], de Nicola and Hennessy [1984], Hoare [1985], and Olderog and Hoare [1986]. Bisimulation-based systems include Abramsky [1991], Austry and Boudol [1984], Baeten and van Glabbeek [1987], Bergstru and Klop [1984, 1985], and Midner [1983, 1984].

The idea of a "silent" (aka "hidden" or " τ-") action plays an important role in both CSP and CCS theories, but creates significant technical and methodological problems; for example, bisimulation ignoring silent actions is not a congruence with respect to the choice operation of CCS. In this paper, we assume for simplicity that *there is no silent action*. Preliminary investigations indicate that our conclusions about bisimulation generally apply to the case with silent moves; this is a matter of ongoing study [Ulidowski 1992].

In the absence of silent action, bisimulation is known to be a congruence with respect to all the operations of CSP/CCS, and Milner has argued that in

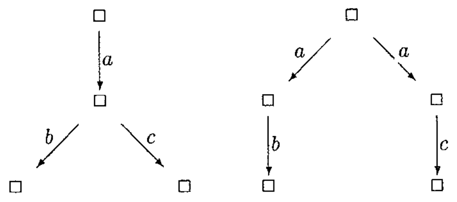

FIG. 1. Trace equivalent but not trace congruent.

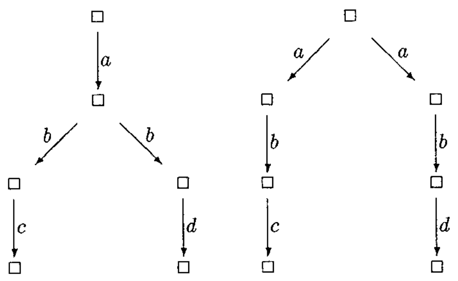

FIG. 2. CSP trace congruent bit not bisimilar.

this case bisimulation yields the finest appropriate notion of the behavior of concurrent processes based on synchronization trees. Although there is some ground for refining synchronization trees further (cf. [Castellano et al. 1987; Plotkin and Pratt 1988]), we shall accept the thesis that bisimilar trees should not be distinguished. Thus, we admit below only operations with respect to which bisimulation remains a congruence. Since bisimilar trees are easily seen to be trace equivalent, it follows in this setting that bisimulation refines all of the trace congruences under consideration. Our results focus on the converse question of whether further identifications should be made, that is, whether nonbisimilar processes are truly distinguishable in their observable behavior.

We noted that a pair of nonbisimilar trees P and Q can be distinguished by an "interactive" protocol. The protocol *itself* can be thought of as a new process Bisim$[P, Q]$. One might suppose that in a general concurrent programming language, it would be possible to define the new process too, and that success or failure of Bisim$[\cdot, \cdot]$ running on a pair P, Q would be easily visible to an observer who could observe traces.

However, CSP and CCS operations are very similar, and the example of Figure 2 shows that bisimulation is a strictly finer equivalence than trace congruence with respect to CSP/CCS operations. It follows that the contexts Bisim$[\cdot, \cdot]$ distinguishing nonbisimilar processes by their traces *are not definable using the standard CSP/CCS operations*; if they were, nonbisimilarity could be reduced to trace distinguishability. Namely, any pair of nonbisimilar trees P and Q would also be trace distinguishable by plugging them in for X in Bisim$[X, P]$ and observing the "success" trace when P is plugged in, but not when Q is plugged in.

Thus, we maintain that implicit in concurrent process theory based on bisimulation is another "interactive" kind of *meta*process, which the formalisms of CSP/CCS—languages proposed as bases for understanding interactive processes—are inadequate to define! The central question of this paper [Abramsky 1987; Bloom et al. 1988] is

What further operations on CCS/CSP terms are needed so that protocols reducing nonbisimilarity to trace distinguishability become definable?

In the remainder of the paper, we argue that bisimulation *cannot* be reduced to a trace congruence with respect to any *reasonably structured* system of process constructing operations. The implications of this conclusion are discussed in the final section.

In particular, we formulate in Section 4.3 a general notion of a system of processes given by structured rules for transitions among terms—a GSOS system.[1] Almost all previously formulated systems of bisimulation-respecting operations are definable by GSOS rule systems; the exception, in Groote and Vaandrager [1989] and later papers, have been explorations of the power of structured operational semantics. Even rules with negative antecedents are allowed in GSOS systems. On the other hand, we indicate in Section 6 that any of the obvious further relaxations of the conditions defining GSOS's can result in systems which are ill-defined, countably branching, or fail to respect bisimulation. Thus, we believe that GSOS definability provides a generous and mathematically robust constraint on what a reasonably structured system of processes might be. We therefore restrict our attention to GSOS operations.

Definition 1.1. Two processes are *GSOS trace congruent* iff they yield the same traces in all GSOS definable contexts.

Our first main result is that bisimulation—even restricted to finite trees—is a *strict* refinement of GSOS trace congruence. Specifically, we develop in Section 7.2 a characterization of GSOS trace congruence similar to the standard characterization of bisimulation and use it to prove:

THEOREM 1.2. *The nonbisimilar trees* $P_\star = a(bc + bd)$ *and* $Q_\star = a(bc + bd) + abc$ *(cf Figure 7) are GSOS trace congruent.*

We remark that GSOS congruence is a strict refinement of CSP congruence. A map of the equivalences used in this paper is given in Figure 3; the higher equivalences are finer than the lower ones. More detail on the collection of process equivalences based on synchronization trees can be found in Abramsky [1987, 1989] and Abramsky and Vickers [1989].

THEOREM 1.3. *The processes* $aa + ab$ *and* $aa + ab + a(a + b)$ *(see Figure* 4) *are CSP trace congruent (de Nicola and Hennessy* [1984], *axiom* (D5), *p.* 99), *but not GSOS trace congruent.*

GSOS congruence has a theory comparable to that of bisimulation in its richness. In Section 7.2, we demonstrate that GSOS congruence is equivalent to *ready simulation*, a relation between processes whose formulation closely resembles bisimulation. In Section 8, we present a modal characterization of ready simulation similar to that of Hennessy and Milner [1985], and in Section 9, we show that it is the trace congruence with respect to two simple GSOS operations and CSP.

Abramsky [1987] independently raised the question of how to test distinguishability of nonbisimilar processes and formalized the operational behavior of a set of protocols that do capture bisimulation. In Bloom [1989] and Bloom et al. [1990], we offer a similar system, slightly improved in certain respects. As

[1] Originally, the "G" in "GSOS" stood for "guarded recursion." We argue in Section 4.2 that guarded recursion is an inessential feature for our purposes. However, the acronym has been used in two many places by too many authors to be easily changed. It now might as well stand for "Grand."

Synchronization Tree Isomorphism	$P \equiv Q$	The finest acceptable process equivalence in this theory. We insist that processes with the same tree be identified, and that all languages respect tree isomorphism. This relation is *too fine*; we want the non-isomorphic processes a and $a + a$ to be identified.
Bisimulation	$P \underline{\leftrightarrow} Q$	Generally accepted in this community to be the *finest* acceptable process equivalence; that is, if P and Q are bisimilar, then P and Q ought to be considered identical. Solves the a vs. $a{+}a$ problem of tree isomorphism. In this study we argue that bisimulation is too fine.
Ready Simulation (GSOS Congruence) ($\frac{2}{3}$-bisimulation)	$P \rightleftharpoons Q$	Congruence with respect to all well-structured languages. In this paper, we present an introduction to the theory of ready simulation, suggesting that it is a mathematically appealing alternative to bisimulation, and unlike bisimulation makes only computationally meaningful distinctions.
Trace Congruence	$P \equiv_{tr}^{\mathcal{L}} Q$	Trace equivalent in all $\mathcal{L}$-contexts.
CSP Congruence (Failures Equivalence)	$P \equiv_{tr}^{\mathrm{CSP}} Q$	Congruence with respect to CSP. This enters our discussion only incidentally.
Trace Equivalence (Automaton Equivalence)	$P \equiv_{tr} Q$	P and Q have the same finite completed traces. Not compositional for several basic operations such as CCS restriction or CSP parallel composition.

FIG. 3. Equivalences used in this study.

the main result of this paper requires, both systems use ill-structured features in essential ways.

2. *Preliminaries*

There are several possible ways to understand CCS; we take a fairly denotational approach. Terms are a notation for *synchronization trees*, which are

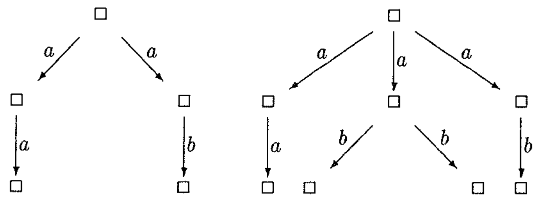

FIG. 4. CSP trace congruent but not ready similar.

essentially infinite-state loop-free automata. We use some nonempty, finite set Act of *actions*, which are not given any further interpretation. We use a, b, c as metavariables ranging over actions. Unlike Milner's original CCS [Milner 1983], our general theory does not require any algebraic structure on the set of actions; the algebraic structure can be encoded in natural ways if desired. Note that we are trying to work in as finite a setting as possible; our action set, unlike Milner's, must be finite.[2]

Definition 2.1. A *synchronization tree* is a rooted, unordered, finitely branching, possibly infinitely deep tree with edges labeled by elements of Act. A *countably-branching synchronization tree* is a rooted, unordered, possibly countably wide and deep tree with Act-labeled edges. We call synchronization trees "finitely branching" when we wish to emphasize the distinction between finitely and countably branching trees.

Definition 2.2. The tree P' is a $\left\{\begin{matrix}\text{child}\\ a\text{-child}\\ \text{descendant}\\ s\text{-descendant}\end{matrix}\right\}$ of P if there is $\left\{\begin{matrix}\text{an arc}\\ \text{an arc labeled } a\\ \text{a path}\\ \text{a path labeled } s\end{matrix}\right\}$ from the root of P to the root of P', where $a \in \mathsf{Act}$ and $s \in \mathsf{Act}^*$. If P' is an a-child (respectively, s-descendant) of P, we write $P \overset{a}{\to} P'$ (respectively, $P \overset{s}{\to} P'$).

A set Δ of trees is *downward closed* if whenever $P \in \Delta$, all descendants of P are in Δ.

It is clear how to consider a tree as an infinite-state nondeterministic automaton; the root of the tree is the start state of the automaton, and on each step it selects an edge and performs the action labeling that edge. Conversely, given such an automaton, it can be unwound into a (finitely or countably branching) synchronization tree in an obvious way.[3]

2.1. BISIMULATION. Bisimulation is a pure mathematical notion; it is independent of the language in question. Despite this, we will see that it is an adequate semantics for any well-structured language. To give a hint of the motivation of bisimulation (and because it is useful in later work) we mention an even finer adequate semantics first.

Definition 2.1.1. If P and Q are synchronization trees, $P \equiv Q$ if P and Q are isomorphic as unordered edge-labeled trees.

[2]As discussed in more detail in Section 4.2, we are interested in *distinguishing* processes rather than expressing them; distinctions between processes should be observable with a finite amount of computation; in any reasonable setting, this will use only a finite number of actions.
[3]Trees are used because they simplify certain definitions. Further, most core languages (without recursion) give a notation for all finite trees, but not for all finite automata.

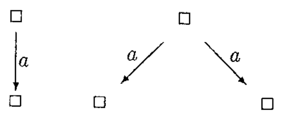

FIG. 5. a and $a + a$.

Although the synchronization tree semantics $\equiv$ of CCS is adequate and simple to think about, it is not the right semantics. Consider the two processes a and $a + a$ of Figure 5. Each can only perform a single a-action and then stop; $a + a$ can do so in two ways. All a-actions are the same, and all stopped processes act the same—so there should be no distinction between a and $a + a$. Indeed, there is general agreement that synchronization trees are an overspecification of process behavior, and certain trees must be regarded as equivalent. The question, then, is which trees to identify. In CCS, Milner chose to identify precisely the *bisimilar* trees.

As befits a good mathematical notion, there are several equivalent definitions of bisimulation.[4] Properties of bisimulation have been elaborated in Hennessy and Milner [1985] and Milner [1984]; we present only selected highlights.

Definition 2.1.2. A relation $\sim$ between synchronization trees is a *strong bisimulation relation* if, whenever $P \sim Q$ and $a \in \mathsf{Act}$, then

—whenever $P \xrightarrow{a} P'$, then for some $Q', Q \xrightarrow{a} Q'$ and $P' \sim Q'$, and, symmetrically,
—whenever $Q \xrightarrow{a} Q'$, then for some $P', P \xrightarrow{a} P'$ and $P' \sim Q'$.

For example, synchronization tree isomorphism and the null relation are both strong bisimulation relations. So is $\sim_0$, where $P \sim_0 Q$ iff P and Q are isomorphic, or if P and Q are isomorphic to a and $a + a$, respectively.

Definition 2.1.3. P and Q are bisimilar, written $P \leftrightarroweq Q$, iff there is a strong bisimulation relation $\sim$ such that $P \sim Q$.

For example, $\sim_0$ shows that $a \leftrightarroweq a + a$. Indeed $P \leftrightarroweq P + P$, where $P + P$ is two copies of P joined at the root. The relation $\leftrightarroweq$ is itself a strong bisimulation relation, and in fact the largest one. It is an equivalence relation, and even a congruence relation with respect to all the operations of CCS.

One of the other characterizations of bisimulation from Hennessy and Milner [1985] will be particularly important for this study. The following logical characterization holds only for finitely branching trees. It can be extended to trees with larger branching, at the cost of introducing infinitary conjunctions and disjunctions.

Definition 2.1.4. The *Hennessy–Milner* [1985] *formulas* over Act are inductively defined as:

—*tt* and *ff*
—$\varphi \wedge \psi$ and $\varphi \vee \psi$
—$\langle a \rangle \varphi$ for each $a \in \mathsf{Act}$.
—$[a]\varphi$ for each $a \in \mathsf{Act}$.

[4]Throughout this paper, "bisimulation" is Milner's "strong bisimulation."

If φ is a Hennessy–Milner formula and P is a finitely-branching synchronization tree, then the relation of *satisfaction*, $P \models \varphi$, is defined by:

—$P \models tt$ always, $P \models ff$ never;
—$P \models (\varphi \wedge \psi)$ iff $P \models \varphi$ and $P \models \psi$, and similarly for disjunction;
—$P \models \langle a \rangle \varphi$ iff $P' \models \varphi$ for some P' such that $P \xrightarrow{a} P'$.
—$P \models [a]\varphi$ iff $P' \models \varphi$ for every P' such that $P \xrightarrow{a} P'$.

For example, $P \models \langle a \rangle tt$ iff P has an a-child. If $\varphi = \langle a \rangle [b] \langle c \rangle tt$, then φ separates the processes of Figure 7: $a(bc + bd) + abc \models \varphi$ but $a(bc + bd) \not\models \varphi$. The familiar laws of propositional logic hold for Hennessy–Milner formulas, and so we ambiguously write, that is, $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ knowing that the order of parenthesization is irrelevant. Hennessy and Milner [1985] show the following:

THEOREM 2.1.5. *If P_1 and P_2 are finitely-branching synchronization trees, then $P_1 \leftrightarrow P_2$ iff $P_1 \models \varphi$ iff $P_2 \models \varphi$ for each Hennessy–Milner formula φ.*

In particular, bisimulation is fully abstract with respect to observing modal formulas. That is, if we have some way of testing $P \models \varphi$ for all P and φ, then we have a good reason to distinguish between non-bisimilar trees. However, it is hard to see how to observe modal formulas without observing too much [Bloom and Meyer 1992], or to understand them as computational observations.

3. *Setting*

3.1. SIGNATURES AND TRANSITION RELATIONS. We will be studying the interaction of programming languages and semantics, and in particular we will vary the programming language. We therefore give general definitions suitable for quantifying over languages. In general, we want to have the most powerful class of languages that can be achieved without losing the essential mathematical and aesthetic properties that characterize CCS. We propose a class of languages, the GSOS languages, and argue that it meets this goal.

A language in the style of CCS is given by a *signature* (a set of operation symbols), and an operational semantics over that signature. It is convenient to include a set of synchronization trees in each language, so that we can test two trees for equivalence in different languages without having to worry about actually defining them by terms in the two languages.[5] We will include the nodes of the trees themselves as constants in the language.

Definition 3.1.1. A *signature* $\mathscr{F}$ is a nonempty finite set of *actions* $\mathsf{Act}_{\mathscr{F}}$, a possibly empty downward-closed set $\Delta_{\mathscr{F}}$ of synchronization trees over $\mathsf{Act}_{\mathscr{F}}$, and a family of disjoint sets $\mathscr{F}_i$ for $i = 0, 1, 2, \ldots$ such that $\bigcup_{i \in \mathbf{N}} \mathscr{F}_i$ is finite. The elements of $\mathscr{F}_i$ are the *operation symbols of arity i*. We insist that $\mathbf{0} \in \mathscr{F}_0$, $\mathsf{Act}_{\mathscr{F}} \subseteq \mathscr{F}_1$, and $+ \in \mathscr{F}_2$.

[5]We will frequently want to include *all* synchronization trees. For pedantic reasons, we only include only a few representatives from each of the 2^{ω} isomorphism classes of synchronization trees, but we ignore this subtlety from now on.

We fix an infinite set of Var of *variables*, called X, Y, Z which will range over process terms. The set of open terms over $\mathscr{F}$ is the least set such that

—Each synchronization tree $P \in \Delta_{\mathscr{F}}$ is a term.
—Each $X \in$ Var is a term
—$f(P_1, \ldots, P_k)$ is a term whenever $f \in \mathscr{F}_k$ and each P_i is a term.

A *closed* term is a term that contains no variables, as we have no binding operators (see Section 4.2). Substitution $P[\vec{X} := \vec{Q}]$ is defined in the standard way.

Aside from the required $\mathbf{0}$, $a(\cdot)$, and $+$, CCS has the binary operations $|$ and $\|$, and the unary operations $\backslash L$ for $L \subseteq$ Act and $[\rho]$ for certain ρ: Act $\rightarrow$ Act. Similar languages such as MEIJE [Austry and Bondal 1984], CSP, and ACT use fairly similar sets of operations. We will not be programming in CCS per se, so we will not give the full semantics for CCS. In ordinary discourse, operations are written in prefix, infix, or suffix form as appropriate.

The operations are generally given meaning by *structured operational rules* [Plotkin 1981]; a language for concurrency in the CCS/CSP style is completely defined by a signature together with a set of structured rules defining the relation $P \xrightarrow{a} Q$ on closed terms. The operational semantics are given as a labeled transition system on the set of closed terms, which we will unwind into a synchronization tree in the obvious way.

It is difficult to define "structured operational rule" in sufficient generality to cover all the ways used in the literature (e.g., Barendregt [1981; 1984], Bloom [1988], Meyer [1988], Plotkin [1981]) and simultaneously avoid pathologies in particular cases. The results in this paper, among other work, show that the properties of a system defined by structured rules can be quite sensitive to the form of the rules. In general, though, a structured rule has the form

$$\frac{\textit{antecedent}}{\textit{consequent}},$$

where the anticedent and consequent are statements that may have free variables. These variables are considered bound in the rule, and rules that differ only in the names of free variables are identified; for example, the following are identical:

$$\frac{x \xrightarrow{a} y}{f(x) \xrightarrow{b} g(y)} \quad \frac{y \xrightarrow{a} x}{f(y) \xrightarrow{b} g(x)}.$$

The intent of the rule is that whenever the antecedent is satisfied by some instantiation of the free variables, then so is the consequent; and conversely that whenever a fact holds, there should be some instantiation of some rule in the language with true antecedent and that fact as consequent. Structured rules, in a variety of guises, are familiar in many areas of mathematics, computer science, and logic.

We will frequently use rule schema in a fairly informal way; for example, the following scheme describes two rules for each $a \in$ Act.

$$\frac{\textit{antecedent}[a](a \in \mathsf{Act})}{\textit{consequent-}1, \textit{consequent-}2}.$$

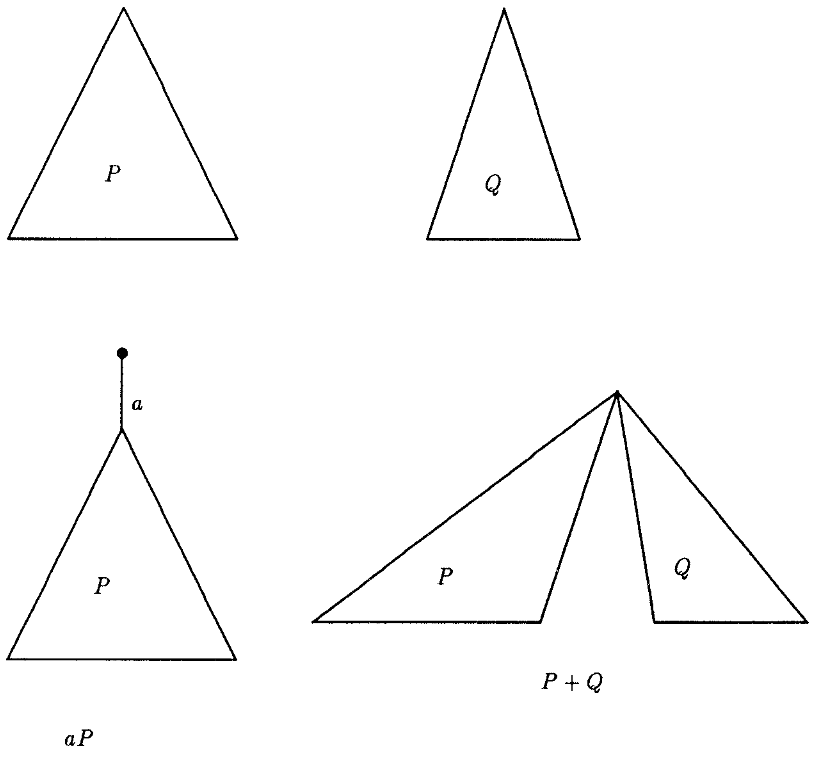

FIG. 6. Effects of basic operations on trees.

We illustrate the informal use of rules by giving rules for the required operations. We require that all languages have the same rules for these operations. The full definitions will be given in Section 4.3.

—If P is a synchronization tree and P' is an a-child of P, then $P \xrightarrow{a} P'$.
—$\mathbf{0}$ has no rules; it denotes the null tree.
—For each $a \in \mathsf{Act}$, we have the following rule.[6] The synchronization tree of aP is that of P with a new root and an edge labeled a from the new root to the root of P.

$$aX \xrightarrow{a} X \tag{2}$$

—For each $a \in \mathsf{Act}$, we have the following two rules. The synchronization tree of $P + Q$ consists of the trees of P and Q with roots identified.

$$\frac{X \xrightarrow{a} Y}{X + X' \xrightarrow{a} Y,\ X' + X \xrightarrow{a} Y} \tag{3}$$

See Figure 6 for pictures of aP and $P + Q$. We write a for the process $a\mathbf{0}$ when no confusion can arise, use infix notation, and omit parentheses following usual mathematical conventions. For example, we mercifully write $a(bc + bd)$

[6]We could make a distinction between axioms and interference rules; for our purposes, it is simplest to consider axioms such as this as interference rules with an empty set of hypotheses.

for $a(+(b(c(\mathbf{0})), b(d(\mathbf{0}))))$. It is easy to show that $+$ is commutative and associative in the synchronization-tree semantics—that is, for all synchronization trees P and Q, $P + Q$ and $Q + P$ are isomorphic as synchronization trees. We have terms denoting all finite synchronization trees: $\mathbf{0}$ denotes the tree with no actions, and if P_i denotes the tree t_i, then $a_1 P_1 + \cdots + a_n P_n$ denotes the tree with an a_i-edge to t_i for each $1 \leq i \leq n$.

Let P be a closed term of $\mathscr{F}$, and $\longrightarrow$ a transition relation over $\mathscr{F}$. The behavior of P under $\longrightarrow$ is a possibly infinite graph edge-labeled by actions, and node-labeled by terms.[7] This graph may be unwound to give a possibly infinite tree $[\![P]\!]\longrightarrow$; if it is finitely branching, it is a synchronization tree giving the meaning of P in an obvious sense. Thus, given a transition relation, definitions phrased in terms of synchronization trees may be applied to processes as well; we write, for example, $P \leftrightarrow Q$ for $[\![P]\!]\longrightarrow \; \leftrightarrow [\![Q]\!]\longrightarrow$.

More generally, given an open term P of variables $X_1, \ldots, X_n$, we may interpret P as an n-ary function on synchronization trees. For example, aX denotes the function of prepending a new root and an a-branch to the original root of a tree. Examples of processes and their associated synchronization trees can be found in most of the figures in this article.

3.2. Observations. As stated in the introduction, we are mainly interested in *finite completed traces* or simply *traces*: sequences of actions that processes may take before they halt. This is formalized in the following definition.

Definition 3.2.1. If $s \in \mathsf{Act}^*$, then $P \xrightarrow{s} P'$ iff there are terms $P_1, \ldots, P_{|s|}$ such that

$$P \xrightarrow{s_1} P_1 \xrightarrow{s_2} \cdots \xrightarrow{s_{|s|}} P_{|s|} = P'.$$

We write $P \xrightarrow{s}$ if for some P' we have $P \xrightarrow{s} P'$; otherwise, $P \overset{s}{\nrightarrow}$. P is *stopped* if for every $a \in \mathsf{Act}$, $P \overset{a}{\nrightarrow}$.

It is also possible to use *partial traces*, strings s such that $P \xrightarrow{s}$, or *infinite traces*, infinite strings s such that there exist P_i's such that $P_0 = P$ and $P_i \xrightarrow{s_{i+1}} P_{i+1}$ for each i. Partial traces are too weak for our purposes, and infinite traces may justifiably be regarded as impossible to observe. In this study, "trace" will always mean "finite terminated trace" unless otherwise specified.

We have used the notation $P \xrightarrow{a} Q$ in two ways for synchronization trees P and Q, in the senses of Definition 2.2 and Definition 3.2.1 the two notions are equivalent on all synchronization trees.

Definition 3.2.2. The *trace set* of P, $\mathrm{tr}(P)$, *is* $\{s \in \mathsf{Act}^* | P \xrightarrow{s} P'$ and P' is stopped$\}$.

We formalize "using a program" by the notion of a context.

[7]To do this in full detail, one should formalize the notion of a "proof of $P \xrightarrow{a} Q$", and have one a-edge from P to Q for each proof of $P \xrightarrow{a} Q$. With the straightforward definition, the term $a + a$ would have the same synchronization tree as a; however, the term $a(\mathbf{0} + \mathbf{0}) + a\mathbf{0}$ would have a different synchronization tree: in particular, the synchronization tree semantics would not be adequate. Counting proofs corrects this anomaly. This subtlety is irrelevant for our purposes, and we do not pursue it.

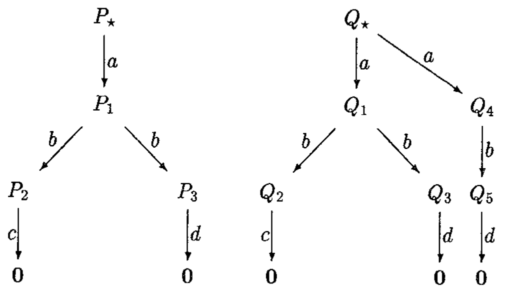

FIG. 7. $P_\star$ and $Q_\star$: Ready similar but not bisimilar.

Definition 3.2.3. A *context of n holes* $C[X_1, \ldots, X_n]$ in a language $\mathscr{L}$ is simply an $\mathscr{L}$-term with free variables at most $\{X_1, \ldots, X_n\}$. The result of simultaneously substituting P_i for X_i in $C[X_1, \ldots, X_n]$ is written $C[P_1, \ldots, P_n]$.

For example, $C[X] = (X + a)|(X + b)$ is a CCS context. There are no variable-binding operations available in our language, so no issues of variable renaming arise in performing substitutions.

Definition 3.2.4. Let P and Q be synchronization trees.

(1) *P trace approximates Q*, $P \sqsubseteq_{tr} Q$, iff $tr(P) \subseteq tr(Q)$.
(2) P and Q are *trace equivalent*, $P \equiv_{tr} Q$, iff $tr(P) = tr(Q)$; that is, if $P \sqsubseteq_{tr} Q \sqsubseteq_{tr} P$.
(3) *P trace approximates Q* with respect to the language $\mathscr{L}$, $P \sqsubseteq_{tr}^{\mathscr{L}} Q$, iff for all $\mathscr{L}$-contexts $C[X]$ of one free variable, $C[P] \sqsubseteq_{tr} C[Q]$.
(4) P and Q are *trace congruent with respect to the language* $\mathscr{L}$, $P \equiv_{tr}^{\mathscr{L}} Q$, iff for all $\mathscr{L}$-contexts $C[X]$ of one free variable, $C[P] \equiv_{tr} C[Q]$.

In CSP, then, two programs are distinguished iff there is a good reason to distinguish them—a context $C[X]$ and a string s of actions such that only one of $C[P]$ and $C[Q]$ can perform s and then stop. In fact, there is a fully abstract mathematical semantics of CSP, the *failures* semantics of Brookes et al. [1984]. It is well-known that bisimulation is strictly finer than failures semantics. Logically, failure semantics correspond to modal formulas of the form

$$\langle a_1 \rangle \langle a_2 \rangle \cdots \langle a_k \rangle (([b_1]\mathit{ff}) \wedge ([b_2]\mathit{ff}) \wedge \cdots \wedge ([b_l]\mathit{ff}));$$

that is, two processes are CCS/CSP congruent iff they agree on all such formulas. From this, it is routine to check that $P_\star \equiv_{tr}^{\mathrm{CCS/CSP}} Q_\star$ but $P_\star \not\leftrightarrow Q_\star$ (see Figure 7).

One might wonder why $P_\star$ and $Q_\star$ should be considered different in CCS. In the spirit of Abramsky [1987] and Plotkin [1977], we investigate the question of what kinds of operations one must add to CCS to make bisimulation be trace congruence.

4. *GSOS Languages*

4.1. GENERALIZING CCS—A FALSE START. As we have seen, bisimulation is not trace congruence with respect to CCS. We consider various ways of generalizing CCS, attempting to refine the language's congruence to make it coincide with bisimulation.

There is a straightforward and trivial operation to add to CCS that makes bisimulation precisely equal trace congruence. This operation, called "not-bisim," takes two arguments; it produces a signal if they are not bisimilar, and is stopped if they are bisimilar.

$$\frac{X \not\leftrightarrow X'}{\text{not-bisim}(X, X') \xrightarrow{a} \mathbf{0}}.$$

In this language, if P and Q are not bisimilar, then the context $C[X] =$ not-bisim(X, P) distinguishes them: $C[P] \overset{a}{\nrightarrow}$, but $C[Q] \xrightarrow{a} \mathbf{0}$. This operation may be criticized.

—It begs the question. We have explained why we think bisimulation is important by saying that, if we consider it important, then it is important. Any other relation between synchronization trees could be explained in the same way. For example, a slight variation on that rule would make synchronization tree isomorphism the fundamental relation in the language, distinguishing a from $a + a$.

—The rule is very difficult to apply. In Milner's original SCCS, the question of $P \leftrightarrow Q$ is not arithmetic [Bloom 1987]. Even for finitely-branching trees, the question $P \not\leftrightarrow Q$ is r.e.-complete, and so the one-step transition relation $P \xrightarrow{a} Q$ is not decidable. It is semidecidable,[8] which is why we phrased the rule in terms of $\not\leftrightarrow$ rather than $\leftrightarrow$.

We would like to choose some criterion by which we may judge languages and see if they are reasonable. We will argue that bisimulation is not trace congruence with respect to any reasonable language; to make this formal, we will have to have some way of quantifying over all reasonable languages, and hence need some formal characterization of those languages.

One of the elegant features of CCS is the definition by a set of structured operational rules. By contrast, not-bisim is defined by an ad-hoc rule involving the predicate of bisimulation. From the form of the structured rules, CCS is guaranteed to exhibit such useful mathematical properties as that all programs using only guarded recursion are finitely branching, and that the transition relation is decidable. We choose, therefore, to investigate languages defined by rules which look like the rules of CCS. We check the soundness of our definition of "looking like the rules of CCS" by making sure that we maintain the essential properties of CCS. We will try to catch all reasonable languages by taking the largest cleanly defined class of languages with "well-structured" CCS-like definitions that have these essential properties. The remainder of this section is concerned with the discussion of our proposed class of "reasonable" CCS-like process languages.

4.2. THE PURPOSE OF FIXED POINTS. We are investigating the *discriminatory* power of the language, its ability to tell the difference between processes in finite time. The fixed point operations add to the *expressive* power, the ability of the language to define synchronization trees and functions on them. In general, the two kinds of power are related: Increasing the discriminatory power necessarily increases the expressive power. However, the converse does

[8]Strictly, the question $P \not\leftrightarrow Q$ is r.e. relative to the tree constants appearing in P and Q. If there are no nonrecursive trees, then the question is r.e..

not hold: It is common to discover that a language extension has increased expressive power and left discriminatory power unchanged. Most programmers and language designers are quite properly concerned with expressive power—the ability to write programs easily—and at most secondarily interested in discriminatory power. However, we are working on issues of full abstraction, in which discriminatory power is paramount.

One programming-language feature commonly included in core languages for concurrency is *recursion*. Using recursion requires some caution. It is possible to define processes (e.g., fix $[X.a + (X \| X)]$), which are *infinitely branching*; that is, there are a countable set of distinct terms Q_n such that fix $[X.a + (X \| X)] \xrightarrow{a} Q_n$. Infinitely branching trees and so-called "unguarded processes" [Milner 1983] cause problems in many aspects of the theory. Unguarded recursion can make the one-step transition relation $P \xrightarrow{a} Q$ undecidable, suggesting that the language may be theoretically intractible and undesirably hard to implement or model-check. The correspondence between bisimulation and Hennessy–Milner logic becomes harder; Theorem 2.1.5 fails, unless infinitary conjunctions and disjunctions are added.

For these reasons and others, restrictions are generally imposed on recursive definitions of processes. In CCS, "guarded" recursion is singled out as attractive, and in CSP, and the test-equivalence system of de Nicola and Hennessy [1984], unguarded recursions are treated as though they diverged. The essence of these restrictions is to ensure that definable processes behave like computable, finitely branching trees: that there is a program which, given a and P, computes the finite set $\{Q | P \xrightarrow{a} Q\}$.

In the case of guarded recursion, suppose that P and Q are not trace congruent—that is, there is a context $C[X]$ and a string s of actions such that, say, $s \in \mathrm{tr}(C[P]) - \mathrm{tr}(C[Q])$. This context $C[X]$ may involve recursion. However, the guarded fixed point operators appearing in $C[X]$ may be unwound a suitably large but finite number of times and then replaced by $\mathbf{0}$ giving a new context, $C'[X]$, without fixed point operators and also has the property that $s \in \mathrm{tr}(C'[P]) - \mathrm{tr}(C'[Q])$. That is, P and Q can be distinguished by a context not involving recursion at all. This informal argument can be formalized directly in our class of languages.

Thus, we do not include the recursion operator in the class of GSOS languages. It may be noted that GSOS languages include the *expressive* power of recursion, in the sense that any set of guarded recursive definitions over a GSOS language $\mathcal{L}$ can be added to $\mathcal{L}$ as new constants, and the extended language is still GSOS. This is roughly equivalent to the recursion principle of many modern process algebras (e.g., Milner et al. [1992]).

4.3. GSOS RULES. We present the general format of *GSOS structured transition rule*, and then argue that none of the obvious restrictions in this format may be dropped. The argument will take the form of theorems showing that any language defined by GSOS rules has desirable properties, and counterexamples showing that the desirable properties are lost in the obvious extensions of the GSOS format.

Definition 4.3.1. 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 $T \overset{a}{\nrightarrow}$.

Definition 4.3.2. A *GSOS rule* ρ is a rule of the form:

$$\frac{\bigcup_{i=1}^{l}\left\{X_i \xrightarrow{a_{ij}} Y_{ij} | 1 \le j \le m_i\right\} \cup \bigcup_{i=1}^{l}\left\{X_i \overset{b_{ik}}{\not\rightarrow} | 1 \le k \le n_i\right\}}{\mathsf{op}(X_1, \dots, X_l) \xrightarrow{c} C\left[\vec{X}, \vec{Y}\right]},$$

where all the variables are distinct, $l \ge 0$ is the arity of op, $m_i, n_i \ge 0$, 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.) The operation symbol op is the *principal operator* of the rule; $\mathsf{ante}(\rho)$ is the set of antecedents, and $\mathsf{cons}(\rho)$ is the consequent.

A rule is *negative* if it has any $x_i \overset{b_{ij}}{\not\rightarrow}$ antecedents; otherwise, it is *positive*. Note that every X_i occurring in the antecedent of a GSOS rule must occur as an argument of the principal operator in the consequent, but not every argument of the principal operator need occur in the antecedent.

Definition 4.3.3. A *GSOS rule system* $\mathcal{G}$ over a signature $\mathcal{F}$ is a finite set of GSOS rules over the actions and operations in $\mathcal{F}$, such that precisely the rules (2) and (3) are given for the operations $a(\cdot)$ and $+$.

We first show that each GSOS rule system determines an operational semantics. The operational semantics will be given by a labeled transition system with the closed $\mathcal{F}$-terms as the processes and the actions in Act as the actions. The presence of negative rules requires additional care defining the transition relation.

Definition 4.3.4. A (closed) substitution is a partial map σ from variables to (closed) terms. We write $P\sigma$ for the result of substituting $\sigma(X)$ for each X occurring in P; if $\sigma(X)$ is undefined, $P\sigma$ is undefined.

For example, let $\sigma(X) = Q$ and $Y \notin \mathrm{dom}(\sigma)$; then $(aX + bY)\sigma = aQ + bY$. Note that if the free variables in P are $X_1, \dots, X_n$, then $P\sigma = P[\vec{X} := \vec{Q}]$. All substitutions in this study will be closed.

Definition 4.3.5. Let $\rightsquigarrow$ be a transition relation, σ be a substitution, and t be a transition formula. The predicate $\rightsquigarrow, \sigma \models t$ is defined by

$$\rightsquigarrow, \sigma \models T \xrightarrow{a} T' \Leftrightarrow T\sigma \overset{a}{\rightsquigarrow} T'\sigma,$$

$$\rightsquigarrow, \sigma \models T \overset{a}{\not\rightarrow} \Leftrightarrow \nexists\, Q.T\sigma \overset{a}{\rightsquigarrow} Q.$$

If $\mathcal{S}$ is a set of transition formulas, $\rightsquigarrow, \sigma \models \mathcal{S}$ iff $\forall t \in \mathcal{S}.\ \rightsquigarrow, \sigma \models t$.

For example, let $\rightsquigarrow_{\mathrm{CCS}}$ be the transition relation of CCS (which we write as $\rightarrow$ in all sections in which the notation is not ambiguous). Suppose that $\sigma_1(X) = ab + ac$. Then, we have $\rightsquigarrow_{\mathrm{CCS}}, \sigma_1 \models \{X \xrightarrow{a} b, X \xrightarrow{a} c, X \overset{b}{\not\rightarrow}\}$.

Definition 4.3.6. *If ρ is a GSOS rule, $\rightsquigarrow, \sigma \models \rho$ holds iff*

$$\rightsquigarrow, \sigma \models \mathsf{ante}(\rho) \textit{ implies } \rightsquigarrow, \sigma \models \mathsf{cons}(\rho).$$

For example, $\rightsquigarrow_{\mathrm{CCS}}, \sigma \models \rho$ for every substitution σ and every CCS rule ρ. However, we also have $\rightsquigarrow_{\infty}, \sigma \models \rho$ for every CCS rule as well, where $\rightsquigarrow_{\infty}$ is the universal relation: $P \overset{a}{\rightsquigarrow}_{\infty} Q$ for all P, Q, and a.

Definition 4.3.7. $\rightsquigarrow$ is *sound* for $\mathcal{G}$ iff for every rule $\rho \in \mathcal{G}$ and every substitution σ, we have $\rightsquigarrow, \sigma \models \rho$.

In general, many transition relations will be sound for $\mathcal{G}$; for example, $\rightsquigarrow_{\infty}$ is sound for every $\mathcal{G}$. One generally takes the *smallest* sound transition relation, showing that there is in fact a smallest one. This is not appropriate for GSOS rules; with negative rules, there may not be a smallest sound transition relation [Bloom 1989].

However, GSOS languages do define a (unique) operational semantics in an appropriate sense. The point of minimality in the usual case is to ensure that everything that is true is true for some reason, because there is some rule with that fact as consequent and a true antecedent. We make this concern explicit.

Definition 4.3.8. $\rightsquigarrow$ is *witnessing* for $\mathcal{G}$ iff, whenever $P \overset{a}{\rightsquigarrow} P'$ there is a rule $\rho \in \mathcal{G}$ and a substitution σ such that $\rightsquigarrow, \sigma \models \mathsf{ante}(\rho)$ and $\mathsf{cons}(\rho)\sigma = P \xrightarrow{a} P'$.

A transition relation is witnessing if, whenever a transition happens, it happens because it was the consequent of (an instantiation of) some rule, and the antecedents of that rule were satisfied. For example, $\rightsquigarrow_{\mathrm{CCS}}$ is witnessing for CCS. However, $\rightsquigarrow_{\infty}$ is not witnessing for CCS. There are no axioms for $\mathbf{0}$, yet $\mathbf{0} \overset{a}{\rightsquigarrow}_{\infty} a$. Soundness and witnessing together select the right transition relation:

LEMMA 4.3.9. *Let $\mathcal{G}$ be a GSOS rule system. There is a unique sound and witnessing transition relation $\rightarrow_{\mathcal{G}}$ for $\mathcal{G}$.*

PROOF. Straightforward structural induction. □

We call the unique transition relation the *standard* transition relation, and write it $\rightarrow_{\mathcal{G}}$ or simply $\rightarrow$.

5. *Why GSOS Rules Are Desirable*

In this section, we argue that GSOS rules are appropriate as a generalization of CCS. We give two theorems that, together with Lemma 4.9, demonstrate that bisimulation is appropriate in the GSOS setting. Recall that bisimulation is best used with finitely branching trees; we will show that every GSOS language produces only finitely branching trees. We then give an indication of the additional power of GSOS-definable operations by some examples of operations on trees that can be defined in the GSOS setting but not in CCS.

5.1. BASIC PROPERTIES

THEOREM 5.1.1. *Let $\mathcal{G}$ be a GSOS rule system. Then the transition relation on $\mathcal{G}$ is computably finitely branching uniformly in the tree constants. That is, there is an algorithm that, given an action a, a term P, and oracles for all the tree constants occurring in P, produces the (necessarily finite) set of a-children of P.*

PROOF. A straightforward recursion on P. □

As desired, all GSOS operations respect bisimulation

THEOREM 5.1.2. *Let $\mathcal{G}$ be a GSOS rule system. Then bisimulation is a congruence with respect to the operations in $\mathcal{G}$. That is, if $P \leftrightarrow Q$ are synchronization trees and $C[X]$ is a context over $\mathcal{G}$, then $C[P] \leftrightarrow C[Q]$.*

PROOF. This is similar to the proof of Lemma 7.8, presented in Section 7.4; it is best done using the machinery developed in that section. □

COROLLARY 5.1.3. *If $P \leftrightarrow Q$, then P and Q are trace congruent with respect to $\mathcal{G}$.*

PROOF. It is clear that, if $R \leftrightarrow S$, then R and S have the same traces. Let $C[\cdot]$ be an arbitrary context; by Theorem 5.1.2, $C[P] \leftrightarrow C[Q]$, and hence P and Q have the same traces in $C[\cdot]$. □

5.2. EXPRESSIVE POWER. GSOS rules are quite expressive, as witnessed by the fact that most structured transition rules proposed in the field have been GSOS rules. For example, the CCS restriction operations $P \restriction A$ for $A \subseteq \mathsf{Act}$ are defined by the family of rules, one for each $a \in A$:

$$\frac{X \xrightarrow{a} Y}{X \restriction A \xrightarrow{a} Y \restriction A}.$$

The simple interleaving parallel composition, $\|$, is given by:

$$\frac{X \xrightarrow{a} Y}{X \| X' \xrightarrow{a} Y \| X'}, \quad \frac{X' \xrightarrow{a} Y}{X \| X' \xrightarrow{a} X \| Y} \tag{4}$$

(with one instance of each rule for each action a.) The standard parallel composition operation $|$ of CCS has these rules, and some extra rules for communication. Suppose that there is a distinguished action $\tau \in \mathsf{Act}$, and a permutation $\bar{\cdot}$ of $\mathsf{Act} - \{\tau\}$, such that $\bar{\bar{a}} = a$ for each action $a \neq \tau$. The remaining behavior of $|$ is given by the rule scheme

$$\frac{X \xrightarrow{a} Y,\ X' \xrightarrow{\bar{a}} Y'}{X | X' \xrightarrow{\tau} Y | Y'}.$$

The operational rules assigning behavior to CCS/CSP/ACP/MEIJE terms easily fit the GSOS framework.

In fact, GSOS rules go beyond the kind of structured rules needed for CCS in two aspects—the use of *negation* and *copying*. Negation allows us to define an elemental form of sequential composition: $(P; Q)$ runs P until it stops, and then runs Q. As an operation on synchronization trees, $(P; Q)$ gives a copy of Q at each leaf of P.[9]

$$\frac{X \xrightarrow{a} Y}{X; X' \xrightarrow{a} Y; X'} \quad \frac{X' \xrightarrow{b} Y',\ \left\{X \overset{a}{\not\rightarrow} \mid a \in \mathsf{Act}\right\}}{X; X' \xrightarrow{b} Y'}.$$

[9]It is possible to define some forms of sequential composition with positive rules. For example, sequencing in CSP runs P until it announces that it has finished by sending a special action, then runs Q. However, processes may finish without announcing that they have finished—called "deadlock" in CSP—or (in general) may announce that they have finished when in fact they are still able to continue; CSP sequencing is not identical with pure sequencing. It is evidently acceptable for programming.

Copying allows us, not surprisingly, to make copies of processes. There can be more than one antecedent about the behavior of a single subprocess, and more than one copy of a process in the result in the consequent. For example, the following GSOS rules yield operations that cannot be defined in CCS [de Simone 1985].

$$\frac{X \xrightarrow{\mathtt{t}} Y}{\mathtt{while}\, X\, \mathtt{do}\, X' \xrightarrow{\mathtt{t}} X';\, \mathtt{while}\, Y\, \mathtt{do}\, X'} \qquad \frac{X \xrightarrow{a} Y}{!X \xrightarrow{a} Y \| !X}.$$

The `while` operation is the heart of an utterly standard while-loop; it simply needs some rules allowing the test X to interact with the outside world. The ! operation [Milner 1990] turns X into a reentrant server, allowing it to spawn off new X's as needed. Both of these operations are implementable—in many settings, with no copying in the implementation. Like any semantics, a GSOS definition of a language should be regarded as a specification, not a guide for implementation.

6. *Obvious Extensions Violate Basic Properties*

There are many technical restrictions in our definition of a GSOS rule, and it is natural to ask if they can be relaxed. We indicate how various relaxations may break the key properties of GSOS systems. Note that some systems with non-GSOS rules enjoy the good properties of GSOS systems; however, this is not immediate from the syntactic specifications of these systems. GSOS rules therefore provide a language-design methodology: Any language defined purely by GSOS is guaranteed to meet the basic criteria; other languages may or may not. Perhaps more importantly, a GSOS language may be extended by GSOS operations and is still guaranteed to behave well; a well-behaved non-GSOS language extended by the same GSOS operations may cease to be well behaved. The properties that non-GSOS systems most often violate are:

—The guarantee that bisimulation is a congruence. In fact, they typically do not respect synchronization tree isomorphism; for example, there are two stopped programs that can be distinguished. (Recall that all stopped programs are unable to take any actions, and hence they have the same synchronization tree; in fact, the null tree.)

—The requirement that $\xrightarrow{\cdot}$ be computably, finitely branching.

—The existence of some transition relation $\xrightarrow{\cdot}$ agreeing with all the rules.

Many possible extensions of the GSOS format give some kind of pattern-matching ability, which generally prevents bisimulation from being a congruence. For example, the consequent must be of the form $\mathsf{op}(\vec{X}) \xrightarrow{c} C[\vec{X},\vec{Y}]$. If we allow more structured left-hand sides of the consequent, we allow a certain kind of pattern matching. Consider a unary operation ζ defined by the axiom

$$\zeta(\mathbf{0}) \xrightarrow{a} \mathbf{0}.$$

Now, $\zeta(\mathbf{0}) \xrightarrow{a} \mathbf{0}$ but $\zeta(\mathbf{0} + \mathbf{0})$ is stopped. This gives us a context which distinguishes between the two bisimilar terms $\mathbf{0}$ and $\mathbf{0} + \mathbf{0}$, so bisimulation is not a congruence with respect to ζ. A series of similar counterexamples are given in Appendix A.

7. *Theory of Ready Simulation*

7.1. Overview. In this section, we develop the core of the theory of ready simulation and GSOS languages. We present and prove the equivalence of the main definitions of ready simulation, and in particular we show that ready simulation is precisely congruence with respect to all GSOS languages. We also develop a modal logic which matches ready simulation in the same way that Hennessy-Milner logic matches bisimulation. In Section 9, we use this logic to build a GSOS language in which ready simulation is precisely congruence. As a corollary, we show that bisimulation is not congruence with respect to any GSOS language.

The three main characterizations presented in this chapter are ready simulation (RS), denial logic (DL), and GSOS congruence (GC). We prove the equivalences in the order

$$\begin{tikzcd}
\mathrm{DL} \arrow[rr, leftrightarrow] \arrow[dr, leftrightarrow] & & \mathrm{RS}. \\
& \mathrm{GC} \arrow[ur, leftrightarrow] &
\end{tikzcd}$$

It is simpler to talk about synchronization trees (which are absolute) rather than process terms (which change their meaning depending on the language.)

Definition 7.1.1. Let P and Q be synchronization trees.

(1) $P \sqsubseteq_{tr}^{\mathrm{GSOS}} Q$ iff, for all GSOS languages $\mathcal{G}$ including P and Q as trees, $P \sqsubseteq_{tr}^{\mathcal{G}} Q$.

(2) $P \equiv_{tr}^{\mathrm{GSOS}} Q$, P and Q are *GSOS congruent*, iff for all GSOS languages $\mathcal{G}$ including P and Q as trees, $P \equiv_{tr}^{\mathcal{G}} Q$.

Two processes in the language $\mathcal{G}$ are GSOS congruent iff their synchronization trees with respect to $\mathcal{G}$ are.

7.2. Ready Simulation and GSOS Congruence. The following characterization was discovered by Larsen and Skou [1991], and independently by van Glabbeek [1993].

Definition 7.2.1

(1) A relation $\sqsubseteq'$ between synchronization trees is a *ready simulation* relation iff, whenever $P \sqsubseteq' Q$,
—whenever $P \xrightarrow{a} P'$, then there is a Q' such that $Q \xrightarrow{a} Q'$ and $P' \sqsubseteq' Q'$.
—whenever $P \overset{a}{\nrightarrow}$, then $Q \overset{a}{\nrightarrow}$.

(2) $P \underset{\rightarrow}{\sqsubset} Q$ if there is some ready simulation relation $\underset{\rightarrow}{\sqsubset}'$ such that $P \underset{\rightarrow}{\sqsubset}' Q$.

(3) $P \rightleftharpoons Q$ iff $P \underset{\rightarrow}{\sqsubset} Q$ and $Q \underset{\rightarrow}{\sqsubset} P$. In this case P and Q are said to be *ready similar*.

A useful fact follows immediately from the definition. Let the *ready set* of P be defined by

$$\mathrm{readies}(P) = \left\{ a : P \xrightarrow{a} \right\}. \tag{5}$$

Then, $P \underset{\rightarrow}{\sqsubset} Q$ implies readies(P) = readies(Q). In the presence of the first bullet in the definition of ready simulation, readies(P) = readies(Q) is equiva-

$P_\star$	$\underset{\sim}{\sqsubset}_1$	$Q_\star$	$Q_\star$	$\underset{\sim}{\sqsubset}_2$	$P_\star$
P_1	$\underset{\sim}{\sqsubset}_1$	Q_1	Q_1	$\underset{\sim}{\sqsubset}_2$	P_1
P_2	$\underset{\sim}{\sqsubset}_1$	Q_2	Q_2	$\underset{\sim}{\sqsubset}_2$	P_2
P_3	$\underset{\sim}{\sqsubset}_1$	Q_3	Q_3	$\underset{\sim}{\sqsubset}_2$	P_3
			Q_4	$\underset{\sim}{\sqsubset}_2$	P_1
			Q_5	$\underset{\sim}{\sqsubset}_2$	P_3
0	$\underset{\sim}{\sqsubset}_1$	0	0	$\underset{\sim}{\sqsubset}_2$	0

FIG. 8. Ready simulation relations.

lent to the second clause. The name "ready simulation" comes from the use of the set of actions that the process is ready to perform.

The relation $\underset{\sim}{\sqsubset}$ is a ready simulation relation, and in fact the largest such relation. The main result of this section is that $P \rightleftharpoons Q$ iff P and Q are GSOS congruent. Proving this will take the rest of the section. Before proving it, we give some examples.

7.3. EXAMPLES OF READY SIMULATION. For any process P, we have $P \underset{\sim}{\sqsubset} P$. Furthermore, for any P and Q we have $aP \underset{\sim}{\sqsubset} aP + aQ$, using the relation $\underset{\sim}{\sqsubset}$ itself: for example, $bc \underset{\sim}{\sqsubset} bc + bd$. The only possible transition of aP is $aP \xrightarrow{a} P$, and $aP + aQ \xrightarrow{a} P$ and $P \underset{\sim}{\sqsubset} P$ as desired. This inequality, together with the axioms of bisimulation, gives a complete inequational axiom system for ready simulation of finite trees. The canonical example of processes that are ready similar but not bisimilar are $P_\star = a(bc + bd)$ and $Q_\star = abc + a(bc + bd) + abd$ of Figure 7; the ready simulations relation between them are given in Figure 8.

Note that a bisimulation relation is a ready simulation relation in each direction, and so bisimilar processes are ready similar. We therefore have:

THEOREM 7.3.1. *Bisimulation is a strict refinement of ready simulation and hence of GSOS congruence.*

A final example of two processes that are ready similar but not bisimilar are *lossy delay links*. A lossy two-stage link repeatedly accepts an input value v, waits one time unit, and produces as output either the value v or a signal $\mho$ saying that v was lost in transit. We present two ways to specify the lossy delay link in CCS. The first always receives its input correctly, but may lose it during the delay; the second may lose it either on the step that it receives it or during the delay. We use the action d for delay, and $v?$ and $v!$ for the input and output of the value v.

$$LL_1 = \sum_v v?.(d.v!.LL_1 + d.\mho.LL_1)$$

$$LL_2 = \sum_v (v?.(d.v!.LL_2 + d.\mho.LL_2) + v?.d.\mho.LL_2).$$

Frequently, one wishes to show a protocol is correct even if the communication medium may lose messages; LL_1 and LL_2 are two ways to code this. They are GSOS congruent, and hence interchangeable. However, they are not bisimilar; it is thus possible that a protocol could be correct up to bisimulation using one of them as the communication medium, but incorrect for the other. This problem does not arise with GSOS congruence.

7.4. Ready Simulation Implies GSOS Congruence. In this section, we show the following theorem:

Theorem 7.4.1. *Let P and Q be synchronization trees.*

(1) *If $P \sqsubseteq\!\!\!\!\rightarrow Q$, then $P \sqsubseteq_{\mathrm{tr}}^{\mathrm{GSOS}} Q$.*
(2) *If $P \rightleftharpoons Q$, then $P \equiv_{\mathrm{tr}}^{\mathrm{GSOS}} Q$.*

The proof of (1) occupies the rest of the section; (2) follows immediately from (1). Recall that the operations $\mathsf{op}(\vec{X})$ were defined by rules of the form

$$\frac{X_i \xrightarrow{a_{ij}} Y_{ij},\ X_i \overset{b_{ik}}{\nrightarrow}}{\mathsf{op}(\vec{X}) \xrightarrow{c} C\left[\vec{X}, \vec{Y}\right]}.$$

In fact, the same is true of each context $D[\vec{X}]$ as well as the simple contexts $\mathsf{op}(\vec{X})$. That is, the behavior of $D[\vec{X}]$ can be completely captured by a set of derived rules of the form:

$$\frac{X_i \xrightarrow{a_{ij}} Y_{ij},\ X_i \overset{b_{ik}}{\nrightarrow}}{D\left[\vec{X}\right] \xrightarrow{c} C\left[\vec{X}, \vec{Y}\right]}.$$

We will call these constructs "ruloids" rather than "rules" because they are not the rules used to define the language and because they violate our definition of a GSOS rule (the source of the consequent has the wrong form).

Definition 7.4.2. A set of ruloids R is *specifically witnessing* for a context $D[\vec{X}]$ and action c iff all the consequents of ruloids in R are of the form $D[\vec{X}] \xrightarrow{c} C[\vec{X}, \vec{Y}]$, and whenever $D[\vec{P}] \xrightarrow{c} Q$, there is a ruloid $\rho \in R$ and substitution σ such that $\mathsf{cons}(\rho)\sigma = D[\vec{P}] \xrightarrow{c} Q$ and $\rightarrow, \sigma \models \mathsf{ante}(\rho)$.

In the course of the following proof, we will use the notions of "sound and specifically witnessing" at a variety of types—for example, concerning sets of ruloids, or functions returning ruloids. We sketch the definitions where appropriate; but they are essentially the same as Definition 7.4.2.

Theorem 7.4.3. *Let $\mathcal{G}$ be a GSOS language. For each $\mathcal{G}$-context $D[\vec{X}]$ and action a, there exists a finite set $\mathcal{R}(D, a)$ of ruloids of the form*

$$\frac{X_i \xrightarrow{a_{ij}} Y_{ij},\ X_i \overset{b_{ik}}{\nrightarrow}}{D\left[\vec{X}\right] \xrightarrow{a} C\left[\vec{X}, \vec{Y}\right]}$$

such that the rules in $\mathcal{R}(D, a)$ are sound and the set $\mathcal{R}(D, a)$ is specifically witnessing for $D[\vec{X}]$ and a.

PROOF. The proof is by induction on the structure of contexts. If $D[\vec{x}] = x_i$, then the set

$$\mathscr{R}(D, a) = \left\{ \frac{x_i \xrightarrow{a} y}{x_i \xrightarrow{a} y} \right\}$$

clearly suffices.

Suppose, inductively, that $D[\vec{x}] = \mathsf{op}(D_1[\vec{x}], \ldots, D_n[\vec{x}])$. We will construct $\mathscr{R}(D, a)$, using the ruloid sets $\mathscr{R}(D_i, a)$ which exist by the induction hypothesis.

Consider an arbitrary rule ρ with conclusion $\mathsf{op}(\vec{z}) \xrightarrow{a} E[\vec{z}, \vec{w}]$. We will build a set Δ_ρ, a set of ruloids that precisely capture when $D[\vec{P}] \xrightarrow{a} Q$ via rule ρ.

Let $\mathsf{ante}^+(\rho)$ be the set of positive antecedents of ρ, and $\mathsf{ante}^-(\rho)$ the set of negative antecedents.

First, we build the set of antecedents corresponding to $\mathsf{ante}^+(\rho)$. We'd like to replace the antecedent $t = z_i \xrightarrow{a_{ij}} w_{ij} \in \mathsf{ante}^+(\rho)$ by the antecedent $D_i[\vec{x}] \xrightarrow{a_{ij}} w_{ij}$; however, this is not GSOS format. We will instead replace t by the antecedents of a ruloid ρ' giving $D_i[\vec{x}]$ an a_{ij}-transition to $D'_{ij}[\vec{x}, \vec{y}]$, and use $D'_{ij}[\vec{x}, \vec{y}]$ in the place of w_{ij}. We will, of course, have to choose ρ''s in all possible ways for each t.

Let FP_ρ be the set of functions from $\mathsf{ante}^+(\rho)$ to ruloids, such that for each $f \in FP_\rho$ and $t = (z_i \xrightarrow{a_{ij}} w_{ij}) \in \mathsf{ante}^+(\rho)$, $f(t) \in \mathscr{R}(D_i, a_{ij})$, where we rename the target variables y_{ij} if necessary in the $\mathscr{R}(D_i, a_{ij})$ to ensure that they are distinct. Note that if there are no ruloids in $\mathscr{R}(D_i, a_{ij})$ (e.g., if $D_i[\vec{x}] = x_1 \setminus a_{ij}$), then there are no such f's, and our construction will leave Δ_ρ empty. Let

$$A^+_{\rho, f} = \bigcup_{t \in \mathsf{ante}^+(\rho)} \mathsf{ante}(f(t)). \tag{6}$$

Let σ range over substitutions, and let $P_i = \sigma(x_i)$. Then FP_ρ is sound and specifically witnessing for the set of transition formulas $D_i[\vec{x}] \xrightarrow{a_{ij}} w_{ij}$, in the sense that:

(1) If for each i, j, $D_i[\vec{P}] \xrightarrow{a_{ij}} Q_{ij}$, then for some $f \in FP_\rho$ and σ, we have $\sigma \models A^+_{\rho, f}$, and $\mathsf{cons}(f(t))\sigma = D_i[\vec{P}] \xrightarrow{a_{ij}} Q_{ij}$ for each t.

(2) If $\sigma \models A^+_{\rho, f}$, then for each antecedent $t = X_i \xrightarrow{a_i} Y_{ij}$, we have $D_i[\vec{P}] \xrightarrow{a_{ij}} D'_{ij}[\vec{P}, \sigma(y_{ij})]$ where $D_i \xrightarrow{a_{ij}} D'_{ij} = \mathsf{cons}(f(t))$.

The negative antecedents require a bit more work. To translate the antecedent $z_i \overset{b_{ik}}{\nrightarrow}$, we must provide evidence that no ruloid for D_i and b_{ik} can apply to $\vec{P}$. We do this by choosing an antecedent for each ruloid p' in $\mathscr{R}(D_i, b_{ik})$, and asserting its opposite opp(p'). (If some ruloid has no antecedents, then $D_i[\vec{P}] \xrightarrow{b_{ik}}$ always, so rule ρ cannot fire with $D_i[\vec{P}]$ as the ith argument; our construction will leave Δ_ρ empty.)

The opposite of the transition formula $x_i \xrightarrow{a_{ij}} y_{ij}$ of course $x_i \overset{a_{ij}}{\nrightarrow}$. The opposite of $x_i \overset{b_{ik}}{\nrightarrow}$ clearly must have the form $x_i \xrightarrow{b_{ik}} y_{ik}$; however, when we are doing this, we must take care to avoid duplicate uses of variables. The details of the renaming are straightforward but tedious, and we omit them and pretend that $\mathsf{opp}(\cdot)$ is simply a function on formulas.

Let $x_i \overset{b_{ik}}{\nrightarrow}$ be a negative hypothesis of ρ. Let $G_{\rho,i,k}$ be the set of functions g mapping $\mathscr{R}(D_i[\vec{x}], b_{ik})$ to transition formulas, such that for each ruloid $\rho' \in \mathscr{R}(D_i[\vec{x}], b_{ik})$, $g(\rho') \in \mathsf{ante}(\rho')$. Let

$$O_{\rho,g} = \{\mathsf{opp}(g(\rho')) \mid \rho' \in \mathsf{dom}(g)\}. \tag{7}$$

By the inductive hypothesis,

(1) If $\sigma \models O_{\rho,g}$, then $D_i[\vec{x}]\sigma \overset{b_{ik}}{\nrightarrow}$, and

(2) If $D_i[\vec{x}]\sigma \overset{b_{ik}}{\nrightarrow}$, then each rule ρ' must fail because some antecedent $g_{\rho'}$ is not satisfied. Let $g(\rho') = g_{\rho'}$ for all ρ'; then $\sigma \models O_{\rho,g}$.

Let H_ρ be the set of functions h from indices $\langle i, k\rangle$ of negative antecedents, such that $h(i,k) \in G_{\rho,i,k}$. Let

$$A^-_{\rho,h} = \bigcup_{i,k} O_{\rho,h(i,k)}. \tag{8}$$

So, $A^-_{\rho,h}$ gives sufficient conditions for each negative antecedent of ρ to be satisfied; and varying h gives all ways for them to be satisfied.

We finally define

$$\Delta_\rho = \bigcup_{f \in FP_\rho,\, h \in H_\rho} \frac{A^+_{\rho,f} \cup A^-_{\rho,h}}{D[\vec{x}] \xrightarrow{a} E_f[\vec{x}, \vec{y}]}, \tag{9}$$

where $E_f[\vec{x}, \vec{y}] = E[D_i[\vec{x}], D'_{ij}[\vec{x}, \vec{y}]]$, where D'_{ij} is the target of $f(z_i \xrightarrow{a_{ij}} w_{ij})$. From the remarks about the $A^+_{\rho,f}$ and $A^-_{\rho,h}$ constructions, we see that Δ_ρ is sound and specifically witnessing for transitions from $D[\vec{x}]$ by rule ρ.

Finally,

$$\mathscr{R}(D, a) = \bigcup_\rho \Delta_\rho. \tag{10}$$

□

Definition 7.4.4. The *ruloid set* of a GSOS language $\mathscr{G}$ is the union of the sets $\mathscr{R}(D, c)$ given by Theorem 7.4.3.

Clearly, $D[\vec{P}] \xrightarrow{c} P'$ iff there is a ruloid in the ruloid set of $\mathscr{G}$ with consequent of the form $D[\vec{X}] \xrightarrow{c} C[\vec{X}, \vec{Y}]$ specifically witnessing this transition. Now it is fairly straightforward to prove Theorem 7.4.1.

LEMMA 7.4.5. *Let $\mathscr{G}$ be a GSOS language, and trees P and Q in $\mathscr{G}$. If $P \sqsubseteq Q$, then $C[P] \sqsubseteq C[Q]$ for each $\mathscr{G}$-context $C[X]$.*

PROOF. To show that $C[P] \sqsubseteq C[Q]$, it suffices to exhibit a ready simulation relation $\sqsubseteq^\star$ such that $C[P] \sqsubseteq^\star C[Q]$. The obvious candidate is the congruence extension of $\underset{\to}{\sqsubseteq}$ itself, defined by $D[\vec{R}]\underset{\to}{\sqsubseteq}^{\star}D[\vec{S}]$ whenever $\vec{R}$ and $\vec{S}$ are vectors of trees of the right length such that $R_i\underset{\to}{\sqsubseteq}S_i$ for each i. It remains to show that $\underset{\to}{\sqsubseteq}^{\star}$ is a ready simulation relation.

Suppose that $D[\vec{R}] \xrightarrow{c} R'$ for some R'. By Theorem 7.4.3, this is true precisely if there is some ruloid ρ in the ruloid set of $\mathscr{G}$

$$\frac{X_i \xrightarrow{a_{ij}} Y_{ij},\ X_i \overset{b_{ik}}{\nrightarrow}}{D\left[\vec{X}\right] \xrightarrow{c} C\left[\vec{X}, \vec{Y}\right]}$$

and trees R'_{ij} such that $R_i \xrightarrow{a_{ij}} R'_{ij}$, $R_i \overset{b_{ik}}{\nrightarrow}$, and $R' = C[\vec{R}, \vec{R}']$.

As each $R_i\underset{\to}{\sqsubseteq}S_i$, we know that (1) there are S'_{ij} such that $S_i \xrightarrow{a_{ij}} S'_{ij}$ and $R'_{ij}\underset{\to}{\sqsubseteq}S'_{ij}$ for each i, j and (2) $S_i \overset{b_{ik}}{\nrightarrow}$ for each i, k. So, by ρ, we know that $D[\vec{S}] \xrightarrow{c} C[\vec{S}, \vec{S}'] = S'$. By definition of $\underset{\to}{\sqsubseteq}^{\star}$, we know that

$$R' = C\left[\vec{R}, \vec{R}'\right]\underset{\to}{\sqsubseteq}^{\star} C\left[\vec{S}, \vec{S}'\right] = S'$$

and so we have verified the first half of the definition of a ready simulation relation.

The second half is similar; if $D[\vec{R}]$ is unable to take a c-step, then some hypothesis of each ruloid that could allow it to take a c-step must fail. From the fact that $R_i\underset{\to}{\sqsubseteq}S_i$, we discover that the corresponding hypothesis of each ruloid fails for $D[\vec{S}]$ as well, and so $D[\vec{S}] \overset{c}{\nrightarrow}$ as desired. □

This completes the proof of Lemma 7.4.5. To finish Theorem 7.4.1, we must show:

LEMMA 7.4.6. *For all synchronization trees P and Q, if $P\underset{\to}{\sqsubseteq}Q$, then $P\underset{\to}{\sqsubseteq}_{tr}Q$.*

PROOF. Suppose that $P\underset{\to}{\sqsubseteq}Q$ and $P \xrightarrow{s} P'$ with P' stopped. There is a sequence of processes $P = P_0, P_1, \dots, P_n = P'$ such that

$$P_0 \xrightarrow{s_1} P_1 \xrightarrow{s_2} \cdots \xrightarrow{s_n} P_n \text{ stopped.}$$

By definition, we have $P\underset{\to}{\sqsubseteq}Q$. From the definition of $\underset{\to}{\sqsubseteq}$, we know that there are processes $Q = Q_0, Q_1, \dots, Q_n$ such that $Q_i \xrightarrow{s_{i+1}} Q_{i+1}$ and $P_i\underset{\to}{\sqsubseteq}Q_i$ for each i. We have

$$Q_0 \xrightarrow{s_1} Q_1 \xrightarrow{s_2} \cdots \xrightarrow{s_n} Q_n.$$

It remains to show that Q_n is stopped. We have $P_n\underset{\to}{\sqsubseteq}Q_n$, and so readies$(P_n)$ = readies(Q_n). However, P_n is stopped, and so readies$(P) = \varnothing$. Therefore Q_n is stopped, and so s is a completed trace of Q as desired. □

Theorem 7.4.1 now follows routinely. Suppose that $P\underset{\to}{\sqsubseteq}Q$, and $C[X]$ is a context in a GSOS language $\mathscr{G}$. We have $C[P]\underset{\to}{\sqsubseteq}C[Q]$ by Lemma 7.4.5 and then $C[P]\underset{\to}{\sqsubseteq}_{\mathrm{tr}}C[Q]$ by Lemma 7.4.6. Hence, $P \sqsubseteq_{\mathrm{tr}}^{\mathscr{G}} Q$. As this holds for all $\mathscr{G}$, we have $P \sqsubseteq_{\mathrm{tr}}^{\mathrm{GSOS}} Q$.

8. *A Modal Characterization of Ready Simulation*

Recall from Theorem 2.1.5 that bisimulation of finitely branching processes coincides with equivalence with respect to Hennessy–Milner formulas. A similar fact holds for ready simulation. The modal logic is useful for some purposes; for example, it characterizes the properties preserved by ready simulation. Also, the modal characterization is mathematically useful; in Section 9, we use the modal characterization to show that ready simulation is precisely GSOS congruence.

The class of *denial formulas* is

$$\varphi ::== tt \mid ff \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle a \rangle \varphi \mid \neg a. \tag{11}$$

The notion of satisfaction is the same for HML formulas and denial formulas, Definition 2.1.4 with the additional clause $P \models \neg a$ iff $P \overset{a}{\nrightarrow}$. Notice that $\neg a$ is equivalent to a restricted use of the $[a]$ modality, viz. $[a]ff$, and we do not allow the full use of this modality. Denial logic is not closed under negation in any sense; for example, the negation of the formula $\langle a \rangle \langle a \rangle tt$ is not a denial formula.

Definition 8.1

(1) $P \sqsubseteq_{\mathrm{DL}} Q$ iff for all denial formulas φ, $P \models \varphi$ implies $Q \models \varphi$.
(2) $P \equiv_{\mathrm{DL}} Q$ iff $P \sqsubseteq_{\mathrm{DL}} Q$ and $Q \sqsubseteq_{\mathrm{DL}} P$.

THEOREM 8.2 [LARSEN AND SKOU 1988, 1991]. *If P and Q are finitely-branching synchronization trees, then*

—$P \underset{\rightarrow}{\sqsubseteq} Q$ *iff* $P \sqsubseteq_{DL} Q$.
—$P \rightleftharpoons Q$ *iff* $P \equiv_{DL} Q$.

PROOF. The second half follows from the first. Suppose that $P \underset{\rightarrow}{\sqsubseteq} Q$. We show that $P \models \varphi$ implies $Q \models \varphi$ by induction on the structure of φ simultaneously for all P and Q.

(1) tt and ff are trivial.
(2) Suppose $P \models \varphi \wedge \psi$. Then, $P \models \varphi$ and $P \models \psi$, and by induction we have $Q \models \varphi$ and $Q \models \psi$ and hence $Q \models \varphi \wedge \psi$ as desired. Disjunctions are similar.
(3) Suppose that $P \models \langle a \rangle \varphi$. Then, there is a P' such that $P \xrightarrow{a} P'$ and $P' \models \varphi$. As $P \underset{\rightarrow}{\sqsubseteq} Q$, there is a Q' such that $P' \underset{\rightarrow}{\sqsubseteq} Q'$ and $Q \xrightarrow{a} Q'$. By induction, $Q' \models \varphi$; hence, $Q \models \langle a \rangle \varphi$.
(4) Suppose that $P \models \neg a$. Then, $P \overset{a}{\nrightarrow}$, and so $Q \overset{a}{\nrightarrow}$; which is to say $Q \models \neg a$.

To prove the converse, we show that $\sqsubseteq_{\mathrm{DL}}$ is a ready simulation relation. Suppose that $P \sqsubseteq_{\mathrm{DL}} Q$.

—Suppose that $P \xrightarrow{a} P'$. We must show that there is some Q' such that $Q \xrightarrow{a} Q'$ and $P' \sqsubseteq_{\mathrm{DL}} Q'$. Suppose for contradiction that there is no a-child Q' of Q such that $P' \sqsubseteq_{\mathrm{DL}} Q'$. Q has a finite number of a-children, $Q_1, \ldots, Q_n$. For each child Q_i, there is a formula ψ_i such that $P' \models \psi_i$ but $Q_i \not\models \psi_i$. Let $\psi = \psi_1 \wedge \cdots \wedge \psi_n$; if there are no children, then let $\psi = tt$. Then, $P' \models \psi$ and so $P \models \langle a \rangle \psi$. However, $Q \not\models \langle a \rangle \psi$, which violates the assumption that $P \sqsubseteq_{\mathrm{DL}} Q$.

—Suppose that $P \overset{a}{\nrightarrow}$. Then, $P \models \neg a$, and so $Q \models \neg a$. This is equivalent to $Q \overset{a}{\nrightarrow}$ as desired.

We have shown that $\sqsubseteq_{\mathrm{DL}}$ is a ready simulation relation, and so $P \sqsubseteq_{\mathrm{DL}} Q$ implies $P \sqsubseteq Q$. □

In fact, the full syntax of denial formulas is not required; disjunctions and *ff* are not necessary. For example, the formulas $\langle a\rangle(\varphi \vee \psi)$ and $(\langle a\rangle\varphi) \vee (\langle a\rangle\psi)$ are logically equivalent. The *essential denial formulas* are given by the syntax:

$$\varphi ::= = tt \mid \varphi \wedge \varphi \mid \langle a\rangle\varphi \mid \neg a.$$

LEMMA 8.3. *$P \equiv_{DL} Q$ iff P and Q agree on all essential denial formulas.*

PROOF. Use the fact that $\langle a\rangle(\varphi \vee \psi)$ and $(\langle a\rangle\varphi) \vee (\langle a\rangle\psi)$ are logically equivalent, and the other rules of modal logic. □

9. *Ready Simulation Can Be Traced*

In this section, we introduce an extension CCSSS of CCS whose congruence is just ready simulation; that is, $P \equiv_{\mathrm{DL}} Q$ iff $P \equiv_{\mathrm{tr}}^{\mathrm{CCSSS}} Q$. We add two operations. $\mathsf{cp}P$ is a copying operator: when P signals that it wants to fork, $\mathsf{cp}P$ forks. $S \rhd P$ is a sort of controlled communication: S runs alone, except that it occasionally allows P the ability to take a step and communicate with it. These operations correspond to the copying and button-pushing operations in the testing scenarios of Bloom and Meyer [1992].

Using these operations, we code denial formulas into contexts and traces, and so understand ready simulation in CCSSS. $C_\varphi[P]$ tests the process P to see if it satisfies φ, producing a characteristic kind of trace iff P satisfies φ.

Formally, we fix several distinct actions. We use o as a sort of "visible silent action;" processes will emit o's while they are operating. The actions c_1 and c_2 are used by processes to signal to the cp operator that they wish to fork. In $S \rhd P$, S uses the d action to signal that it wishes to communicate with P. There is an auxiliary operator $\rhd_1$ used by $\rhd$.

$\mathsf{cp}(P)$ usually does just what P does. However, when P signals that it wants to be forked (by the c_1 and c_2 actions), $\mathsf{cp}(P)$ forks it.

$$\frac{X \xrightarrow{a} X' \ (a \notin \{c_1, c_2\})}{\mathsf{cp}X \xrightarrow{a} \mathsf{cp}X'} \quad \frac{X \xrightarrow{c_1} X_1,\ X \xrightarrow{c_2} X_2}{\mathsf{cp}X \xrightarrow{o} (\mathsf{cp}X_1) \| (\mathsf{cp}X_2)}.$$

$S \rhd P$ usually does just what S does; P is frozen. However, when S signals that it wants to communicate with P (by performing a d-step), $S \rhd P$ unfreezes P and lets it take a step in cooperation with S. This operation needs a bit of control state, telling whether or not P is frozen; we use the $\rhd$ operator when P is frozen, and the $\rhd_1$ operator when P is unfrozen.

$$\frac{S \xrightarrow{a} S' \ (a \neq d)}{S \rhd P \xrightarrow{a} S' \rhd P} \quad \frac{S \xrightarrow{d} S'}{S \rhd P \xrightarrow{o} S' \rhd_1 P}.$$

The operation $\triangleright_1$ does one step of communication and then behaves like $\triangleright$.

$$\frac{S \xrightarrow{a} S',\ P \xrightarrow{a} P'}{S \triangleright_1 P \xrightarrow{o} S' \triangleright P'}.$$

We now define the coding of essential formulas. To make strings of actions easier to read, we write prefixing with a dot: "$d.a.t.S$" instead of "$datS$." Fix two actions t and f, distinct from the previously mentioned actions, which we use for partial success and total failure.

$$\begin{aligned} S_{tt} &= \mathbf{0} \\ S_{\neg a} &= d.a.f \\ S_{\varphi \wedge \psi} &= c_1 S_\varphi + c_2 S_\psi \\ S_{\langle a \rangle \varphi} &= d.a.t.S_\varphi . \end{aligned}$$

The context $C_\varphi[X]$ is defined to be $\mathsf{cp}(S_\varphi \triangleright X)$. $C_\varphi[P]$ will compute, emitting o's while it is working. Each time it processes an $\langle a \rangle$ correctly, it will emit a t. Each time it fails to perform a $\neg a$ correctly, it will emit an f. We show that $P \models \varphi$ iff $C_\varphi[P]$ produces a completed trace with enough t's and no f's.

For example

$$\begin{aligned} C_{\langle a \rangle tt \wedge \langle b \rangle tt}[a+b] &= \mathsf{cp}((c_1.d.a.t + c_2.d.b.t) \triangleright (a+b)) \\ &\xrightarrow{o} \mathsf{cp}(d.a.t \triangleright (a+b)) \| \mathsf{cp}(d.b.t \triangleright (a+b)) \\ &\xrightarrow{o} \mathsf{cp}(a.t \triangleright_1 (a+b)) \| \mathsf{cp}(d.b.t \triangleright (a+b)) \\ &\xrightarrow{o} \mathsf{cp}(t \triangleright \mathbf{0}) \| \mathsf{cp}(d.b.t \triangleright (a+b)) \\ &\xrightarrow{t} \mathsf{cp}(\mathbf{0} \triangleright \mathbf{0}) \| \mathsf{cp}(d.b.t \triangleright (a+b)) \\ &\xrightarrow{o} \mathsf{cp}(\mathbf{0} \triangleright 0) \| \mathsf{cp}(b.t \triangleright_1 (a+b)) \\ &\xrightarrow{o} \mathsf{cp}(\mathbf{0} \triangleright \mathbf{0}) \| \mathsf{cp}(t \triangleright \mathbf{0}) \\ &\xrightarrow{t} \mathsf{cp}(\mathbf{0} \triangleright \mathbf{0}) \| \mathsf{cp}(\mathbf{0} \triangleright \mathbf{0}) \\ &\text{Stopped.} \end{aligned}$$

To illustrate how the testing for $\neg a$ works, consider:

$$\begin{aligned} C_{\neg a}[(a+b)] &= \mathsf{cp}(d.a.f \triangleright (a+b)) \\ &\xrightarrow{o} \mathsf{cp}(a.f \triangleright_1 (a+b)) \\ &\xrightarrow{o} \mathsf{cp}(f \triangleright \mathbf{0}) \\ &\xrightarrow{f} \mathsf{cp}(\mathbf{0} \triangleright \mathbf{0}) \\ &\text{Stopped.} \end{aligned}$$

So, the only trace of $C_{\neg a}[a + b]$ contains an f. However, the only computation of

$$C_{\neg a}[b] = \mathsf{cp}(d.a.f \rhd b) \xrightarrow{o} \mathsf{cp}(a.f \rhd_1 b)$$

gets stuck after performing an o, and contains no f.

Define $\lfloor \varphi \rfloor$ to be the number of $\langle a \rangle$'s occurring in φ; that is:

$$\lfloor tt \rfloor = \lfloor \neg a \rfloor = 0$$
$$\lfloor \varphi \wedge \psi \rfloor = \lfloor \varphi \rfloor + \lfloor \psi \rfloor$$
$$\lfloor \langle a \rangle \varphi \rfloor = 1 + \lfloor \varphi \rfloor.$$

We say that a trace s is φ-happy if it contains exactly $\lfloor \varphi \rfloor$ t's and no f's. A trace is φ-sad if it contains fewer than $\lfloor \varphi \rfloor$ t's, or at least one f.

LEMMA 9.1. *If $P \models \varphi$, then $C_\varphi[P]$ has a φ-happy trace. If $P \not\models \varphi$, then all traces of $C_\varphi[P]$ are φ-sad. Furthermore, no trace of $C_\varphi[P]$ has more than $\lfloor \varphi \rfloor$ t's.*

PROOF. The proof is by induction on φ.

$\varphi = tt$: $C_{tt}[P]$ is stopped for all P, as desired.

$\varphi = \psi \wedge \theta$: $C_{\psi \wedge \theta}[P] = \mathsf{cp}((c_1 S_\psi + c_2 S_\theta) \rhd P)$. As $((c_1 S_\psi + c_2 S_\theta) \rhd P)$ can make both c_1 and c_2 transitions, the cp forks the process:

$$\begin{aligned} C_{\psi \wedge \theta}[P] &= \mathsf{cp}\big((c_1 S_\psi + c_2 S_\theta) \rhd P\big) \xrightarrow{o} \mathsf{cp}(S_\psi \rhd P) \| \mathsf{cp}(S_\theta \rhd P) \\ &= C_\psi[P] \| C_\theta[P]. \end{aligned}$$

The lemma follows from the ordinary properties of sequences and interleaving.

$\varphi = \neg a$:

$$C_{\neg a}[P] = \mathsf{cp}(d.a.f \rhd P) \xrightarrow{o} \mathsf{cp}(a.f \rhd_1 P).$$

If $P \overset{a}{\nrightarrow}$, then $\mathsf{cp}(a.f \rhd_1 P)$ cannot move and the trace is simply the φ-happy trace o. If $P \xrightarrow{a} P'$, then

$$\mathsf{cp}(a.f \rhd_1 P) \xrightarrow{o} \mathsf{cp}(f \rhd P') \xrightarrow{f} \mathsf{cp}(\mathbf{0} \rhd P').$$

In this case $p \xrightarrow{a}$, the trace of $C_{\neg a}[P]$ is oof, which is φ-sad.

$\varphi = \langle a \rangle \psi$:

$$C_{\langle a \rangle \psi}[P] \xrightarrow{o} \mathsf{cp}(a.t.S_\psi \rhd_1 P)$$

Consider any P' such that $P \xrightarrow{a} P'$. (If there are no such P''s, then the process is stuck and the trace is φ-sad as required.)

$$\mathsf{cp}(a.t.S_\psi \rhd_1 P) \xrightarrow{o} \mathsf{cp}(t.S_\psi \rhd P') \xrightarrow{t} \mathsf{cp}(S_\psi \rhd P') = C_\psi[P'].$$

If $P \models \varphi$, then there is a P' such that $P \xrightarrow{a} P' \models \psi$. By the induction hypothesis $C_\psi[P']$ has a ψ-happy trace, and so we have found a φ-happy trace of $C_\varphi[P]$.

If $P \not\models \varphi$, then $P' \not\models \psi$, and so every trace of $C_\psi[P]$ is ψ-sad; thus every trace of $C_\varphi[P]$ that goes through $C_\varphi[P']$ is φ-sad. Every such trace must go through some $C_\psi[P']$, and so every trace of $C_\varphi[P]$ must be φ-sad. Verifying that no trace has more than $\lfloor \varphi \rfloor$ t's is routine. □

10. *Summary of Ready Simulation*

Combining the results of the previous sections, we obtain the following set of equivalent characterizations of GSOS congruence.

THEOREM 10.1. *The following are equivalent*:

(1) $P \underset{\sim}{\sqsubseteq} Q$ (*State-correspondence definition*)
(2) $P \sqsubseteq_{tr}^{GSOS} Q$ (*Approximation in all GSOS languages.*)
(3) $P \sqsubseteq_{DL} Q$ (*Approximation with respect to all denial formulas*)
(4) $P \models \varphi$ *implies* $Q \models \varphi$ *for all essential denial formulas* φ.
(5) $P \sqsubseteq_{tr}^{CCSSS} Q$ (*Trace approximation in CCSSS*).

COROLLARY 10.2. *Bisimulation is a strict refinement of ready simulation, and hence of GSOS congruence. In particular, the processes* $P_\bigstar$ *and* $Q_\bigstar$ *are trace congruent with respect to every GSOS language, although they are not bisimilar.*

There are a few other definitions of ready simulation, but they are of less interest. For example, it is possible to define the nth approximant to ready simulation as is done for bisimulation in Milner [1980, 1981]; and then $P \underset{\sim}{\sqsubseteq}_n Q$ for all n iff $P \underset{\sim}{\sqsubseteq} Q$.

11. *Conclusion*

Should bisimulation play a significant role in process theory? It has a rich theory, and a tested methodology for verifying correctness of genuine, nontrivial protocols (see, e.g., Baeten [1990], Milner [1989], and Parrow [1987]). Nevertheless, we find unconvincing the arguments for taking bisimulation as a primitive notion. We maintain that computational distinctions should be made only because of observable differences "at the terminal." Global testing systems and modal logics that reduce bisimulation to such observations do not offer what we regard as a reasonable framework for defining operations on processes. We prefer to regard bisimulation as a mathematical tool that is frequently useful in proving programs correct, rather than a characterization of what correctness should mean.[10]

Ready simulation seems to have the mathematical properties which make bisimulation desirable. Bisimulation has several equivalent but distinct definitions, of which the existence of a bisimulation relation and equivalence with respect to a modal logic and their variants are the most useful. Ready simulation has similar definitions. Moreover, the proofs in this paper indicate that those definitions play the same role for ready simulation as they do for bisimulation; for example, the proof of Theorem 7.4.1 given in Section 7.4 uses the existential definition, and the proof in Section 9 that CCSSS congruence is precisely ready simulation uses the modal characterization. These results, and similar work on other aspects of ready simulation, suggest that the theory of

[10] In other work, B. Bloom has used bisimulation methods to verify a silicon compilation scheme [Weber et al. 1992]. The compiler was correct up to bisimulation, and the correctness proof up to bisimulation was no harder than the proof up to trace congruence, so we proved the stronger theorem.

ready simulation is very similar to that of bisimulation in character and power. Of course, ready simulation is easily justified on observational grounds, while bisimulation (despite its other name of "observational equivalence") is harder to justify.

The larger purpose of this study is to illuminate some of the issues one might wish to consider in the choice of a notion of process equivalence. We chose bisimulation as the focus of our study precisely because it had no obvious computational definition. We have given some indications of the form that a computational justification of a notion of equivalence might take: congruence with respect to some sort of well-structured language. Other forms of justification are certainly possible. However, some justification should be considered for each new notion of program equivalence; otherwise, the notion runs the risk of being unjustifiable in computational terms despite having an elegant and powerful mathematical theory. Finally, we have demonstrated that computational justification need not be incompatible with mathematical elegance.

11.1. Related Work. We have sketched the fragment of the theory of ready simulation appropriate to our discussion. There are other questions that one might wish to answer; the answers generally seem to be fairly pleasing. For example, ready simulation of finite processes has a finite axiomatization as an inequational theory [Bloom 1989]; and there is a $O(mn + n^2)$ algorithm for computing if two n-state, m-transition automata are ready similar [Bloom and Paige 1992].

In general, in computer science, two programs are considered equivalent iff they are *congruent*, that is, iff one can be substituted for the other in any context and no difference is observable. This definition has two parameters: the language over which contexts can be formed, and the differences that can be seen. In this paper, we have been varying the language over GSOS and global testing languages; however, the observations have always been finite completed traces. Elsewhere, we examine other notions of observation [Bloom and Meyer 1992].

Bisimulation seems to require two kinds of knowledge, the knowledge of the *possible* and *necessary* behavior of a process, interleaved arbitrarily. The possible behavior appears quite naturally in nondeterminism; each execution of a process gives a possible behavior. The necessary behavior is harder to observe. Larsen and Skou [1991] have proposed the use of probability, testing a process often enough to observe all possible behaviors with high probability. There is a very strong connection between ordinary and probabilistic bisimulation, and a tantalizing and debatable failure of the use of probability as a mechanism for observing bisimulation, and a tantalizing and debatable failure of the use of probability as a mechanism for observing bisimulation; this is discussed in Bloom and Meyer [1992] and Larsen and Skou [1991].

A variation on the theme of well-structured rules is explored in Groote and Vaandrager [1989]. Vaandrager and Groote allow their languages to be countably and undecidably branching. The so-called *tyft/tyxt* rules are positive rules allowing operators in antecedents; the more paradoxical operations defined in the appendix are not allowed. Ready simulation need not be a congruence with respect to such a language. The appropriate notion is the so-called $\Diamond\Box$ equivalence, viz. equivalence with respect to Hennessy–Milner formulas in which no $\langle a \rangle$ operator occurs within the scope of a $[a]$. This is precisely tyft/tyxt congruence; in particular, bisimulation cannot be understood as congruence with respect to any language in this format.

11.2. Open Problems. Throughout this paper, we have assumed that there is no silent (hidden or "τ") action. We expect that similar results hold when there is a silent action; this remains to be verified. Silent moves make the philosophy as well as the mathematics trickier; it is no longer clear what the right questions are.

The first tricky issue is the role of weak bisimulation (Milner's "observational equivalence"). Strong bisimulation is generally accepted as an equivalence that all languages ought to respect. However, weak bisimulation is not a congruence with respect to the $+$ operation of CCS: a and τa are weakly bisimilar, but $a + b$ and $\tau a + b$ are not. In our study, it was clear that we insist that any GSOS language respect strong bisimulation; the corresponding requirement for weak bisimulation is not clear. There are several possible ways to develop the theory. For example, in CCS, weak bisimulation congruence[11] has a simple characterization as *rooted bisimulation*, or equivalently the congruence closure of weak bisimulation with respect to $+$ [Miller 1984]. We could insist that all operations respect this relation. Several variants of weak bisimulation have been proposed, such as the *branching bisimulation* of Baeten and Weijland [1990], which has better equational properties than weak bisimulation. It is possible to find subsets of the GSOS format which respect weak bisimulation and its relatives [Bloom 1993] though the congruences for these rule formats remain to be characterized. It is strictly coarser than bisimulation, as $P_\star$ and $Q_\star$ are trace congruent and hence weakly trace congruent; but they are not weakly bisimilar.

Even the definition of trace congruence is no longer obvious. In the (oversimplified) theory of this study, there was only one way that a process could not perform a visible action: if it is a stopped process. Given a silent move, there are now more ways that a process can do nothing visible. Consider the processes $\mathbf{0}$ (which does nothing), τ (which computes a while and then does nothing), $\tau(\tau + \tau\tau\tau)$ (which does some more complex computation before doing nothing), τ^ω (which thinks forever about what it should do next), and $\tau(\tau + \tau^\omega)$ (which may or may not think forever). The proper identifications between these processes are not obvious; and even the criteria for choosing between the possible choices are subject to some debate. Van Glabbeek [1993] discusses a vast range of choices.

We allowed negative rules because they gave extra programming power and because our theory could handle them. There are several possibilities for combining negative rules and silent moves. Ignoring them would leave a rich theory; leaving them interpreted as they are now might also give an appropriate theory. More elaborate combinations are probably reasonable as well; for example, the hypothesis $P \overset{a}{\nrightarrow}$ could be interpreted as being satisfied iff the set $\{P'\colon P \overset{\tau^\star}{\to} P'\}$ is finite and $P' \overset{a}{\nrightarrow}$ for each such P', or $P \overset{a}{\nrightarrow} \wedge P \overset{\tau}{\nrightarrow}$ as in Ulidowski [1992].

[11]P and Q are weak bisimulation congruent iff for all contexts $C[\cdot]$, $C[P]$ and $C[Q]$ are weakly bisimilar, that is, if P and Q are congruent considering weak bisimulation to be observable.

Appendix A. Counterexamples for Section 6

In this section, we list more counterexamples showing that the GSOS format cannot be extended in other obvious ways.

(i) We do not allow the use of variables to do pattern-matching. For example, the following rule has two different variables taking a-steps and then becoming the same variable:

$$\frac{X_1 \xrightarrow{a} Y,\ X_2 \xrightarrow{a} Y}{\beta(X_1, X_2) \xrightarrow{a} Y}.$$

The context $C[Z] = \beta(a\mathbf{0}, aZ)$ distinguishes between the bisimilar processes $\mathbf{0}$ and $\mathbf{0} + \mathbf{0}$. Copying—using a source variable X_i more than once in the antecedent—does not do any harmful pattern-matching and is acceptable. Duplicated source variables in the term $\mathsf{op}(X_1, \ldots, X_n)$, however, allow too much pattern-matching to respect bisimulation; for example, the rule $\delta(X, X) \xrightarrow{a} \mathbf{0}$ causes as much trouble here as its cousin does in the λ-calculus.

(ii) We have insisted that the positive hypotheses be $X \xrightarrow{a} Y$, but the negative ones $X \overset{b}{\nrightarrow}$. There is no point to hypotheses $X \xrightarrow{a}$; simply use one of the form $X \xrightarrow{a} Y$ and ignore Y. On the other hand, there is a gain of expressive power to negative hypotheses $X \overset{b}{\nrightarrow} Y$. The gain is all in the wrong direction. It is now easy to write some countably branching processes, but it is hard to see how the power of these rules could be useful.

$$\frac{X \overset{b}{\nrightarrow} Y}{\xi(X) \xrightarrow{a} Y}$$

$\xi(\mathbf{0}) \xrightarrow{a} P$ for every process P; it is thus infinitely branching.

(iii) We gain expressive power if we allow terms rather than simply variables to appear in antecedents. If the terms appear as targets—$X \xrightarrow{a} (Y + \mathbf{0})$—we have too much pattern-matching, and examples similar to the previous ones will show that bisimulation need not be a congruence. Terms appearing as the sources cause a more subtle problem: bisimulation will still be a congruence, but a process may be infinitely branching.[12] As an example, consider the nullary operation ω defined by two rules

$$\omega \xrightarrow{a} \mathbf{0} \qquad \frac{\omega \xrightarrow{a} Y}{\omega \xrightarrow{a} aY}.$$

Then, ω is not finitely branching, for we have:

$$\begin{gathered} \omega \xrightarrow{a} \mathbf{0} \\ \omega \xrightarrow{a} a\mathbf{0} \\ \omega \xrightarrow{a} aa\mathbf{0} \\ \vdots \end{gathered}$$

[12] In fact the resulting congruence is coarser than bisimulation; see Groote and Vaandrager [1989] for details.

This occurs because the operation ω is defined by what amounts to an unguarded recursion: it mentions its own one-step behavior in the definition of its one-step behavior. In some circumstances, this might be a useful sort of operation; for example, we could define $\omega'(X) \equiv \sum_{i=0}^{\infty} a^i X$, a process which dawdles for a finite but unbounded time before performing X.

We could also repair this flaw by allowing terms in antecedents, but not allow any operation to refer to itself either directly or indirectly. The resulting class of languages guarantee the essential properties of GSOS languages. In fact, they have almost exactly the same properties: every operation definable with terms in antecedents is definable without them as well. On the other hand, if terms may appear in antecedents, structural induction is no longer a viable proof method.

(iv) Another possible extension is allowing multi-step antecedents [Groote and Vaandrager 1989]. It is in general inconsistent to have both negative and multi-step antecedents. For example, consider the operators $\cdot/a$, α, and π defined as follows:

$$\frac{X \xrightarrow{a} Y_1 \xrightarrow{a} Y_2}{X/a \xrightarrow{a} Y_2} \tag{12}$$

$$\frac{X \overset{a}{\not\rightarrow}}{\alpha(X) \xrightarrow{a} \mathbf{0}}$$

$$\pi \xrightarrow{a} \alpha(\pi/a) \tag{13}$$

It is not hard to show that there is no arrow relation which agrees with these rules: π/a can move iff it cannot move, for:

$$\begin{aligned} &\pi/a \xrightarrow{a} Z \\ \Leftrightarrow\, &\pi \xrightarrow{a} Y \xrightarrow{a} Z \\ \Leftrightarrow\, &Y = \alpha(\pi/a) \xrightarrow{a} Z \\ \Leftrightarrow\, &\pi/a \overset{a}{\not\rightarrow} \quad \text{and} \quad Z = \mathbf{0} \end{aligned}$$

This is similar to the fact that unguarded recursion and negation do not mix. The program fix $[X.\alpha(X)]$ is a term involving an unguarded recursion that can take an a-step iff it cannot take an a-step.

It is possible to have some syntactic conditions that guarantee finite branching and existence of a transition relation, but they are necessarily global. The operation π amounts to an unguarded recursion; however, there is nothing about the form of rule (13) indicating that it is unguarded. This rule would be perfectly acceptable in a GSOS language, for example; it is turned into an unguarded recursion by the presence of the $\cdot/a$ operator. It is nontrivial to define just what a guarded operator is in the presence of multi-step rules; the operator $\cdot/a$ "unguards" X, and an adequate calculus of guardedness remains to be developed. CSP, which has something like the $\cdot/a$ operator, forbids its use in recursions.

(v) A system with multi-step positive antecedents alone is quite possible; such systems respect bisimulation, and always have a minimal transition relation [Groote and Vaandrager 1989]. Two problems arise for such systems. First, sound and witnessing transition relations other than the unique minimal one may exist; second and far more important, such systems admit countably branching processes.

Consider a GSOS language containing the $/a$ operation above, the single action a, and a constant J with the rule

$$J \xrightarrow{a} J/a. \tag{14}$$

Intuitively, J is a process that performs an a-move and then does whatever it does after it performs an a-move. This intuitive definition does not uniquely determine what J does after its a-move; neither does the formalism of sound and witnessing transition relations.

Consider a proof (used loosely) that $J/a \xrightarrow{a} K$ for some K. It must start with the rule for $/a$ used with $x = J$, and thus have hypotheses $J \xrightarrow{a} J_1$ and $J_1 \xrightarrow{a} K$. J has only one transition, from (14), and hence $J_1 = J/a$. That is, any proof that $J/a \xrightarrow{a} K$ is infinitely long, consisting essentially of repeated proofs of $J/a \xrightarrow{a} K$.

Now, we define two transition relations $\dashrightarrow_f$ and $\dashrightarrow_i$, with $P \dashrightarrow_f Q$ there is a finite proof of this, and $P \dashrightarrow_i Q$ if there is an infinite proof. In particular, $J/a \overset{a}{\not\rightarrow}_f$ but $J/a \xrightarrow{a}_i K$ for all processes K. Both $\dashrightarrow_f$ and $\dashrightarrow_i$ are easily seen to be sound and witnessing; as they disagree on J/a, they are not equal.

We know of no circumstance in which the nonuniqueness of the sound witnessing transition relation causes any difficulties; more likely, this simply reflects the fact that the definitions of soundness and witnessing are tuned for GSOS languages. There is a least transition relation in all cases, viz. $\dashrightarrow_f$, and so a "correct" operational semantics.

(vi) More importantly, even the minimal transition relation is not necessarily finitely branching. Consider the operations $\cdot/a$ of (12), and κ defined by:

$$\kappa(X) \xrightarrow{a} X + \kappa(X)/a/a.$$

Let M_0 be a synchronization tree with an infinite derivation of distinct terms

$$M_0 \xrightarrow{a} M_1 \xrightarrow{a} M_2 \xrightarrow{a} \cdots.$$

Let $N = M_0 + \kappa(M_0)/a/a$; so

$$\kappa(M_0) \xrightarrow{a} N.$$

We will show that N has an infinite number of a-children. Clearly $N \xrightarrow{a} M_1$.

Suppose that

$$N \xrightarrow{a} M_i.$$

We have

$$\kappa(M_0) \xrightarrow{a} N \xrightarrow{a} M_i \xrightarrow{a} M_{i+1}$$

and so

$$\kappa(M_0)/a/a \xrightarrow{a} M_{i+1}.$$

As $N = M_0 + \kappa(M_0)/a/a$, we therefore have

$$N \xrightarrow{a} M_{i+1}.$$

In particular, $N \xrightarrow{a} M_j$ for each $j \geq 1$. As the M_j are all distinct, N has an infinite number of a-children.

(vii) However, the effects of multi-step antecedents are worse than simply infinite branching. The transition relation $\dashrightarrow$ ceases to be decidable. It is straightforward to program Turing machines in a suitable GSOS language, so that $M^{(k)} \xrightarrow{a^{f(k)}b} \mathbf{0}$ if the kth Turing machine halts on blank tape (where $f(k)$ is approximately the number of steps the Turning machine takes), and $M^{(k)}$ takes a-steps forever otherwise. Then $\kappa(M^{(k)})/a/a \xrightarrow{b} \mathbf{0}$ precisely if the kth Turing machine halts on blank tape, which is undecidable. The same example shows that the question $P \xrightarrow{a}$ is undecidable.

ACKNOWLEDGMENTS. We would like to thank Frits Vaandrager and Jan Friso Groote for many helpful email discussions, and some of the tricks used in the proof of Section 9. Kim Larsen and Rob van Glabbeek independently discovered the characterization of ready simulation in Definition 7.2.1 and brought it to our attention, which greatly simplified the proof of Theorem 7.4.1 from our original proof based on the modal characterization. Rob van Glabbeek also pointed out that negative GSOS rules are not necessary to capture ready simulation. We would also like to thank Samson Abramsky, Luca Aceto, Ashvin Dsouza, Georges Lauri, Robin Milner, Jon Riecke, Arie Rudich, and Sam Weber for their several contributions. B. Bloom would also like to thank Vicki Borah for boundless emotional support.

REFERENCES

ABRAMSKY, S. 1987. Observation equivalence as a testing equivalence. *Theoret. Comput. Sci. 53*, 2/3, 225–241.

ABRAMSKY, S. 1989. Tutorial on concurrency. Slides of an invited lecture given at the 16th Annual ACM Symposium on Principles of Programming Languages, (Austin, Tex., Jan. 11–13).

ABRAMSKY, S. 1991. A domain equation for bisimulation. *Inf. Comput. 92*, 161–218.

ABRAMSKY, S., AND VICKERS, S. 1989. Observational logic and process semantics. In preparation.

AUSTRY, D., AND BOUDOL, G. 1984. Algèbre de processus et synchronisation. *Theoret. Comput. Sci. 30*, 1, 91–131.

BAETEN, J. C. M. 1990. Applications of process algebra. In *Cambridge Tracts in Theoretical Computer Science*, vol. 17. Cambridge University Press, New York, NY.

BAETEN, J. C. M., AND VAN GLABBEEK, R. 1987. Another look at abstraction in process algebra. In *Automata, Languages, and Programming: 14th International Colloqium*, T. Ottmann, ed. Lecture Notes in Computer Science, vol. 267. Springer-Verlag, New York. July.

BAETEN, J. C. M., AND WEIJLAND, W. P. 1990. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press.

BARENDREGT, H. P. 1981/1984. The Lambda Calculus: Its Syntax and Semantics. In *Studies in Logic*, vol. 103. North-Holland, Amsterdam, The Netherlands.

BERGSTRA, J. A. AND KLOP, J. W. 1984. Process algebra for synchronous communication. *Inf. Comput. 60*, 1–3, 109–137.

BERGSTRA, J. A., AND KLOP, J. W. 1985. Algebra of communicating processes with abstraction. *Theoret. Comput. Sci. 37*, 1, 77–121.

BLOOM, B. 1987. Computational complexity of bisimulation in SCCS. Unpublished.

BLOOM, B. 1988. Can LCF be topped?: Flat lattice models of the simply typed λ-calculus. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*. Computer Society Press, pp. 282–295.

BLOOM, B. 1984. Ready simulation, bisimulation, and the semantics of CCS-like languages. Ph.D dissertation. Massachusetts Institute of Technology, Cambridge, Mass., Aug.

BLOOM, B. 1993. Structural operational semantics for weak bisimulations. Technical Report TR 93-1373, Cornell Univ., Ithaca, N.Y. (Aug.) (*Theoret. Comput. Sci.*, to appear).

BLOOM, B., ISTRAIL, S., AND MEYER, A. R. 1988. Bisimulation can't be traced (preliminary report). In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages* (San Diego, Calif., Jan 13–15) ACM, New York, pp. 229–239. (Also appears as MIT Tech. Memo MIT/LCS/TM-345. MIT, Cambridge, Mass.)

BLOOM, B., ISTRAIL, S., AND MEYER, A. R. 1990. Bisimulation can't be traced. Tech. Rep. TR 90-1150. Cornell Univ., Ithaca, N.Y., (Aug.).

BLOOM, B., AND MEYER, A. R. 1992. Experimenting with process equivalence. *Theoret. Comput. Sci. 102*, 1 (Nov.), 223–237.

BLOOM, B., AND PAIGE, R. 1992. Computing ready simulations efficiently. In *Proceedings of the North American Process Algebra Workshop '92*. Workshops in Computer Science, Springer-Verlag, New York, pp. 119–134.

BROOKES, S. 1983. A semantics and proof system for communicating processes. Tech. Rep. CMU-CS-83-134. Carnegie-Mellon University, Pittsburgh, Pa.

BROOKES, S. D., HOARE, C. A. R., AND ROSCOE, A. W. 1984. A theory of communicating sequential processes. *J. ACM 31*, 3 (July), 560–599.

CASTELLANO, L., MICHELIS, G. D., AND POMELLO, L. 1987. Concurrency vs interleaving: An instructive example. *Bull. Europ. Ass. Theoret. Comput. Sci. 31* (Feb.) 12–15.

DE NICOLA, R., AND HENNESSY, M. C. B. 1984. Testing equivalences for processes. *Theoret. Comput. Sci. 34*, 2/3, 83–133.

DE SIMONE, R. 1985. Higher-level synchronising devices in MEIJE-SCCS. *Theoret. Comput. Sci. 37*, 3, 245–267.

GROOTE, J. F., AND VAANDRAGER, F. 1989. Structured operational semantics and bisimulation as a congruence (extended abstract). In *Automata, Languages and Programming: 16th International Colloquium*, G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, eds. Lecture Notes in Computer Science, vol. 372. Springer-Verlag, New York, pp. 423–438.

HENNESSY, M. AND MILNER, R. 1985. Algebraic laws for nondeterminism and concurrency. *J. ACM 32*, 1 (Jan.), 137–161.

HOARE, C. A. R. 1978. Communicating sequential processes. *Commun. ACM*, *21*, 8 (Aug.), 666–677.

HOARE, C. A. R. 1985. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall.

LARSEN, K., AND SKOU, A. 1988. Bisimulation through probabilistic testing (preliminary report). Tech. Rep. R 88-16. Institut for Elektroniske Systemer, Aalborg Universitetscenter, Aalborg, Denmark, June.

LARSEN, K. G., AND SKOU, A. 1991. Bisimulation through probabilistic testing. *Inf. Comput. 94*, 1 (Sept.), 1–28.

MEYER, A. R. 1988. Semantical paradigms: Notes for an invited lecture, with two appendices by Stavros Cosmadakis. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, Computer Society Press, Washington, D.C., pp. 236–253.

MILNER, R. *A Calculus of Communicating Systems*. In Lecture Notes in Computer Science, vol. 92. Springer-Verlag, New York.

MILNER, 1981. A modal characterisation of observable machine-behaviour. In *CAAP '81*: *Trees in Algebra and Programming*, *6th Colloquium*, E. Astesiano and C. Böhm, eds. Lecture Notes in Computer Science, vol. 112. Springer-Verlag, New York, pp. 25–34.

MILNER, R. 1983. Calculi for synchrony and asynchrony. *Theoret. Comput. Sci.*, *25*, 3, 267–310.

MILNER, R. 1984. Lectures on a calculus for communicating systems. In *Seminar on Concurrency: CMU, July 9–11, 1984*, S. D. Brookes, A. W. Roscoe, and G. Winskel, eds. Lecture Notes in Computer Science, vol. 197. Springer-Verlag, New York, pp. 197–220.

MILNER, R. 1989. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, New York.

MILNER, R. 1990. Functions as processes. In *Automata, Languages and Programming: 17th International Colloqium*, M. Paterson, ed., Lecture Notes in Computer Science, vol. 443. Springer-Verlag, New York, pp. 167–180.

MILNER, R., PARROW, J., AND WALKER, D. 1992. A calculus of mobile processes I and II. *Inf. Comput. 100*, 1–40 and 41–77.

OLDEROG, E.-R., AND HOARE, C. A. R. 1986. Specification-oriented semantics for communicating processes. *Acta Inf. 23*, 1, 9–66.

PARROW, J. 1987. Verifying a CSMA/CD-protocol with CCS. Tech. Rep. ECS-LFCS-87-18, Laboratory for Foundations of Computer Science. University of Edinburgh, Scotland, Jan.

PLOTKIN, G. D. 1977. LCF considered as a programming language. *Theoret. Comput. Sci. 5*, 3, 223–255.

PLOTKIN, G. 1981. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19. Computer Science Department, Aarhus Univ., Denmark.

PLOTKIN, G., AND PRATT, V. 1988. Teams can see pomsets. Manuscript available as pub/pp.tex by anonymous FTP from Boole.Stanford.EDU, Oct.

PNUELI, A. 1985. Linear and branching structures in the semantics and logics of reactive systems. In *Automata, Languages and Programming: 12th Colloquium*, W. Brauer, ed. Lecture Notes in Computer Science, vol. 194. Springer-Verlag, New York, pp. 15–32.

ULIDOWSKI, I. 1992. Equivalences on observable processes. In *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science* IEEE, New York, pp. 148–161.

VAN GLABBEEK, R. 1993. The linear time-branching time spectrum II: The semantics of sequential processes with silent moves. In *Proceedings of CONCUR '93*. E. Best, ed. Lecture Notes in Computer Science, vol. 715. Springer-Verlag, New York, pp. 66–81 (paper also available by anonymous ftp from Boole.stanford.edu.).

WEBER, S., BLOOM, B., AND BROWN, G. 1992. Compiling joy to silicon: A verified silicon compilation scheme. In *Proceedings of the Advanced Research in VLSI and Parallel Systems Conference*, T. Knight and J. Savage, eds. p. 79–98.

RECEIVED JULY 1990; REVISED OCTOBER 1993; ACCEPTED OCTOBER 1993