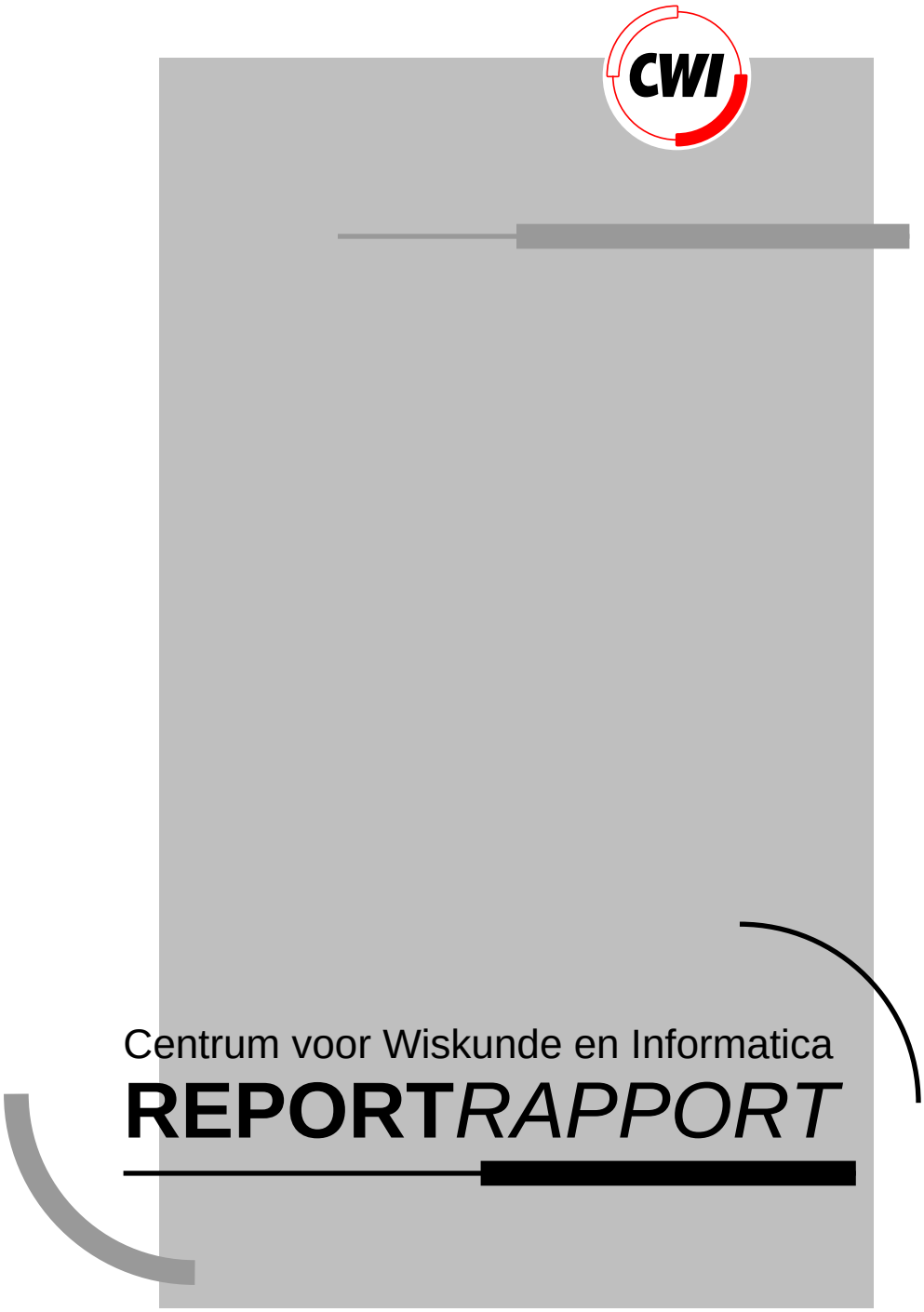

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Expressiveness results for process algebras

F.W. Vaandrager

Computer Science/Department of Software Technology

CS-R9301 1993

Expressiveness Results for Process Algebras

Frits W. Vaandrager
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
fritsv@cwi.nl
University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

Abstract

The expressive power of process algebras is investigated in a general setting of structural operational semantics. The notion of an *effective operational semantics* is introduced and it is observed that no effective operational semantics for an enumerable language can specify all effective process graphs up to trace equivalence. A natural class of Plotkin style SOS specifications is identified, containing the guarded versions of calculi like CCS, SCCS, MEIJE and ACP, and it is proved that any specification in this class induces an effective operational semantics. Using techniques introduced by Bloom, it is shown that for the guarded versions of CCS-like calculi, there is a double exponential bound on the speed with which the number of outgoing transitions in a state can grow. As a corollary of this result it follows that two expressiveness results of De Simone for MEIJE and SCCS depend in a fundamental way on the use of unguarded recursion. A final result of this paper is that all operators definable via a finite number of rules in a format due to De Simone, are derived operators in the simple process calculus PC.

1991 Mathematics Subject Classification: 68Q05, 68Q10, 68Q55, 68Q75, 03D20.

1991 CR Categories: D.3.1, D.3.3, F.1.1, F.1.2, F.3.2, F.4.1.

Keywords & Phrases: process algebra, PC, labeled transition systems, process graphs, effective process graphs, effective operational semantics, structural operational semantics, expressiveness, bisimulation equivalence, trace equivalence, action transducers.

Notes: Most of this work was carried out while the author was at the MIT Laboratory for Computer Science, supported by ONR contract N00014-85-K-0168. Part of this work took place in the context of the ESPRIT Basic Research Action 7166, CONCUR2.

This paper will appear in: J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors. *Proceedings of the REX Workshop "Semantics: Foundations and Applications"*. LNCS, Springer-Verlag, 1993.

1 INTRODUCTION

At this moment there are, besides numerous papers, four introductory textbooks on *process algebra* or, as some prefer to call it, *process theory* by resp. Milner [25], Hoare [21], Hennessy

Report CS-R9301
ISSN
CWI
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands

[20] and Baeten and Weijland [8]. Each of these books gives a thorough introduction into a *particular* approach to process theory. Milner focuses on operational semantics and bisimulation congruences in the setting of his Calculus of Communicating Systems (CCS). Hoare presents his theory of Communicating Sequential Processes (CSP) and concentrates on the denotational failures model. Hennessy elaborates in great detail the notion of testing equivalence for a language somewhere in between CCS and CSP. Baeten and Weijland, finally, advocate the algebraic perspective of the Algebra of Communicating Processes (ACP). A reader who takes the effort to read all the four books, will notice a lot of similarities between the approaches, but will also be puzzled by the differences, and consequently find it hard to make a choice between the available formalisms.

We think that the perspective of the general theory of structural operational semantics can be helpful at this point, because it suggests that the four books just happen to concentrate on different aspects of what can essentially be viewed as a single and homogeneous theory. It is becoming more and more clear that many of the key theorems in process theory are independent of the particular process language that is used. Using Plotkin's structural operational semantics (SOS), one can prove theorems for whole classes of languages at the same time. This is a much more efficient way to develop process theory, which in addition provides more insight. Examples of contributions along these lines are [33, 34, 14, 12, 10, 18, 11, 32, 13, 35, 2, 4, 19].

Milner had the idea that for a proper understanding of the basic issues concerning the behavior of concurrent systems it could be helpful to look for a simple language, with "as few operators or combinators as possible, each of which embodies some distinct and intuitive idea, and which together give completely general expressive power" [24, page 269]. The aim of this paper is to investigate expressiveness issues in a general setting of SOS.

There are at least three different ways in which a language can have "completely general expressive power":

1. Each Turing machine can be simulated in lock step.
2. Each "effective" process graph can be specified up to some notion of behavioral equivalence.
3. Each operation in a "natural" class of operations is realizable in terms of the operations in the language up to some notion of behavioral equivalence.

Most process calculi that have been proposed in the literature are Turing powerful, that is, universally expressive in the first sense.

A first result of this paper, which generalizes a result of Baeten, Bergstra and Klop [6], is that no enumerable language with an effective operational semantics can be universal in the second sense if, as behavioral equivalence, one chooses trace equivalence. Here, two process graphs are called trace equivalent if they have the same finite sequences of actions (so this notion of equivalence does not involve internal actions which can be deleted in a trace). This result implies that if one likes to have a language which is universal in the second sense, one either has to use a notion of behavioral equivalence that does not refine trace equivalence, or one has to give up the idea that the operational semantics should be effective.

A next result of this paper is the definition of a general format of Plotkin style transition system specifications (TSS's), containing the guarded versions of calculi like CCS, SCCS, Meije and ACP, and a proof that any TSS in this class induces an effective operational

semantics. Since (the finitary versions of) process calculi like CCS are effective it follows that these calculi are not universally expressive in the second sense. Also, using techniques introduced by Bloom [10], it is shown that in the guarded versions of CCS-like calculi, there is a double exponential upper bound on the speed with which the fanout, *i.e.*, the number of outgoing transitions in a state, can grow. This implies that there exists a primitive recursive process graph that can not be denoted by CCS-like languages up to trace equivalence.

De Simone [33, 34] proved that any operation on process graphs that can be defined in some general format, can already be defined in SCCS and MEIJE up to bisimulation. As a corollary of the results concerning the growth rate of the fanout, it follows that also this result of De Simone depends in a crucial way on the use of unguarded recursion. The final result of this paper is that a simple calculus called PC is universal in the third sense, that is, each operation definable via a finite number of De Simone style rules, can already be defined in terms of the calculus PC.

ACKNOWLEDGEMENTS. The relational renaming operator of the language PC came up in a discussion with Rob van Glabbeek. Thanks to Jan Bergstra, Doeko Bosscher, Jan Friso Groote and Robert de Simone for useful comments on an earlier version of this paper.

2 A Basic Limitation of Operational Semantics

A *semantics* is a mapping that associates to each object in a syntactic domain a corresponding object in a semantic domain. More specifically, an *operational semantics* is a mapping that associates to each syntactic object a *machine* or *automaton*. These machines (which are mathematical objects) typically have an associated set of *states* and for each state there is a collection of *transitions* which give the possible ways in which the machine can evolve to a next state. This paper takes a rather abstract approach to operational semantics by only considering those aspects of machines and not features like real-time, true concurrency, distribution in space, etc. Thus our machines simply *are* process graphs in the sense defined below.

DEFINITION 2.1 [Process graphs] A *labeled transition system (LTS)* over a given set A of labels is a pair $(S, \longrightarrow)$ where S is the set of *states* and $\longrightarrow \subseteq S \times A \times S$ is the *transition relation*. As usual $r \xrightarrow{a} s$ abbreviates $(r, a, s) \in \longrightarrow$. The *fanout* $fan(s)$ of a state s is defined as the cardinality of the set of transitions starting in s. For $\sigma = a_1 \cdots a_n \in A^*$ a finite sequence over A, predicate $r \xrightarrow{\sigma} s$ is defined by

$$r \xrightarrow{\sigma} s \quad \triangleq \quad \exists r_0, \ldots, r_n \in S : r = r_0 \xrightarrow{a_1} r_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} r_n = s.$$

If $r \xrightarrow{\sigma} s$ for some $\sigma \in A^*$, then state s is called *reachable* from state r.

A *process graph* over A is a triple $g = (r, S, \longrightarrow)$ with $(S, \longrightarrow)$ a LTS over A and $r \in S$ the *root*, such that each state in S is reachable from the root. Sometimes r will be referred to as $root(g)$, and the pair $(S, \longrightarrow)$ as $lts(g)$. If $\mathcal{A}$ is a LTS and s is a state of $\mathcal{A}$, then $graph(s, \mathcal{A})$ is the process graph with root s and an underlying LTS that is obtained by restricting $\mathcal{A}$ to the part that is reachable from s.

Two process graphs g and h are *isomorphic*, notation $g \simeq h$, if there exists a bijective mapping between their sets of states that preserves the roots and the transition relation.

DEFINITION 2.2 [Operational semantics] An *operational semantics* is a mapping that associates to each object in its domain a process graph.

Given our intuition of process graphs as machines that compute, it seems reasonable to focus attention to operational semantics that map expressions to *effective* process graphs, *i.e.*, graphs that have a countable number of states such that in each state the outgoing transitions can be computed. To formalize this notion of effectiveness, we need some simple coding functions known from recursion theory (see [31]). We first introduce a standard coding from ordered pairs of integers to integers:

$$\langle k, l \rangle \triangleq \frac{1}{2} \cdot (k^2 + 2kl + l^2 + 3k + l).$$

The function CI associates to each finite set of integers its *canonical index*, and provides a standard encoding of finite sets of integers into the integers.

$$CI(\{k_1, k_2, \ldots, k_n\}) \triangleq \textbf{if } n = 0 \textbf{ then } 0 \textbf{ else } 2^{k_1} + 2^{k_2} + \ldots + 2^{k_n}.$$

Finally, the function *Gödel* associates to each recursive function ϕ a corresponding Gödel number $G\ddot{o}del(\phi)$.

DEFINITION 2.3 Let $A = \{a_1, a_2, ...\}$ is a enumerable set of actions. A process graph $g = (r, S, \longrightarrow)$ over A is *effective* if

- $S = \{s_1, s_2, \ldots\}$ is an enumerable set of states;
- the transition relation is finitely branching, *i.e.*, for all $s \in S$, $fan(s)$ is finite; and
- the transition relation is effective with respect to the enumerations of S and A. That is, the function $next(g) : \mathsf{N} \rightarrow \mathsf{N}$ defined by

  $$next(g)(i) \triangleq CI(\{\langle k, l \rangle | s_i \xrightarrow{a_k} s_l\})$$

  is recursive.

Graph g is *primitive recursive* if in addition the function $next(g)$ is primitive recursive.

Stated differently, a process graph is effective if there exists a Turing machine that, when provided with a (suitably coded) state as input, computes for a while and then first outputs the number of outgoing transitions from that state and then enumerates all these transitions (everything suitably coded). So in each state it is known what are the possibilities to proceed.

The notion of an effective graph we use here is essentially the same as the one proposed earlier by Baeten, Bergstra and Klop [6] and by Bloom, Istrail and Meyer [12]. A less restrictive definition has been put forward by Darondeau [15], who requires the transition relation, as a set, to be recursive. Boudol [14] and De Simone [34] employ an even less restrictive definition of effectiveness: they only require that the transition relation, as a set, is recursively enumerable.

If the machines, whose behavior is described by means of process graphs, are not in control of all their transitions, then one can argue that our notion of an effective process graph, and in particular the requirement of finite branching, is too restrictive.

Suppose that, like in the I/O automata model of Lynch and Tuttle [23], the set of actions (the labels of transitions) can be partitioned in a set of *input actions*, which are under control of the environment, and a set of *locally controlled actions*, which are under the control of the machine itself. Then it seems reasonable to allow for an infinite number of input transitions from a given state r, provided that, given an input action i, the set of states s which can be reached from r via an i-transition is finite and and can be effectively computed. In each state the machine should be able to decide what to do with a given input.

Also in the case of languages with unbounded nondeterminism due to *random assignment* (see Apt and Plotkin [3]), the requirement of finite branching seems too restrictive.

In this paper, just one particular definition of an effective process graph will be investigated, which certainly is not is the most general one possible.

Baeten, Bergstra and Klop [6] show that, modulo strong bisimulation equivalence, the calculus ACP with finite systems of guarded recursion equations is not universal. Below, we will show that the idea behind the proof of this result can be used to prove a much more general theorem: no effective operational semantics can be universal modulo trace equivalence.

DEFINITION 2.4 An operational semantics $\mathcal{O}$ for an enumerable language L is *effective* with respect to an enumeration $\{p_1, p_2, \ldots\}$ of L if

- for all i, $\mathcal{O}(p_i)$ is effective;
- the function r defined by

  $$r(i) \triangleq \mathit{index}(\mathit{root}(\mathcal{O}(p_i)))$$

  is recursive, where index is the function that associates to each state s_i its index i;
- the function t defined by

  $$t(i) \triangleq \mathit{G\ddot{o}del}(\mathit{next}(\mathcal{O}(p_i)))$$

  is recursive.

An effective operational semantics does not only associate an effective process graph to each expression, but also tells how one can compute the root and transitions of this graph. An example of an operational semantics that is not effective is a mapping that takes a natural number n and associates to it a graph with one state and no transitions if the n-th Turing machine halts, and a graph with one state and one transition otherwise. For each n, the associated graph is finite and hence effective, even though the operational semantics is not.

Let L be a programming language that one likes to implement in accordance with some operational semantics $\mathcal{O}$. If the machines of which $\mathcal{O}$ describes the behavior are in control of all their transitions, then it seems reasonable to require that $\mathcal{O}$ is in fact effective with respect to some enumeration of L. If $\mathcal{O}$ gives no clue about how to build effectively a machine that implements programs in L, then one may even argue that it does not deserve the predicate

"operational". Theorem 2.6 says that, provided this very reasonable requirement is met, there is a limit on what operational semantics can do.

DEFINITION 2.5 For $g = (r, S, \longrightarrow)$ a process graph over A, the set $traces(g)$ is defined by

$$traces(g) = \{\sigma \in A^* \mid \exists s : r \xrightarrow{\sigma} s\}.$$

Process graphs h and h' are *trace equivalent*, notation $h \approx_T h'$, if $traces(h) = traces(h')$.

THEOREM 2.6 *Suppose A is a set of labels containing at least two elements. Suppose $\mathcal{O}$ is an operational semantics that associates to each member of an enumerable language L a process graph over A, and suppose that $\mathcal{O}$ is effective with respect to some enumeration of L. Then there exists an effective graph over A that is not denoted by any member of L up to trace equivalence.*

PROOF Via a diagonalization argument, as in the proof of Theorem 8.2 in [6].
Suppose $\mathcal{O}$ is effective with respect to an enumeration $\{p_1, p_2, \ldots\}$ of L. Let $a, b \in A$ with $a \neq b$. To each $n \in \mathsf{N}$, a function $f_n : \mathsf{N} \to \{0, 1\}$ is associated in the following way:

- $f_n(k) = 0$ if all traces of $\mathcal{O}(p_n)$ of length $k + 1$ end with an action a;
- $f_n(k) = 1$ otherwise.

From the fact that all process graphs $\mathcal{O}(p_n)$ are effective it follows that all f_n are recursive functions. Consequently, the following function $f_\omega : \mathsf{N} \to \{0, 1\}$ is also recursive:

- $f_\omega(n) = 0$ if $f_n(n) = 1$,
- $f_\omega(n) = 1$ if $f_n(n) = 0$.

Now consider the effective process graph Ω with states taken from N, root 0, and transitions

$$n \xrightarrow{a} n + 1 \quad \text{if} \quad f_\omega(n) = 0,$$
$$n \xrightarrow{b} n + 1 \quad \text{if} \quad f_\omega(n) = 1.$$

We claim that, for all n, $\mathcal{O}(p_n) \not\approx_T \Omega$. The proof is by contradiction. Suppose that for some n, $\mathcal{O}(p_n) \approx_T \Omega$. The process graph Ω has exactly one trace of length $n+1$ which either ends with an a or with a b. If it ends with an a, then $f_n(n) = 1$. But this means that there is a trace of $\mathcal{O}(p_n)$ of length $n+1$ that does not end with an a. This contradicts the assumption that $\mathcal{O}(p_n) \approx_T \Omega$. If, in the other case, the unique trace of Ω ends with a b, then $f_n(n) = 0$. But this means that all traces of $\mathcal{O}(p_n)$ of length $n+1$ end with an a, so that again we have a contradiction. ■

Since trace equivalence is coarser than bisimulation equivalence, which in turn is coarser than graph isomorphism, the above theorem has as a trivial corollary that no effective operational semantics can denote all effective graphs modulo bisimulation equivalence or graph isomorphism. In this sense, the above result generalizes Theorem 8.2 of Baeten, Bergstra and Klop [6].

Various researchers have attempted to find universal expressiveness results for languages with an operational semantics in terms of process graphs. They all had to face the limitations imposed by Theorem 2.6, but came up with different solutions:

1. Baeten, Bergstra and Klop [6] prove that each effective process graph can be specified in the language ACP_τ with guarded recursion, modulo an equivalence called weak bisimulation congruence. This universality result is possible because weak bisimulation equivalence is incomparable with trace equivalence, due to the fact that it abstracts from internal actions.

2. De Simone [33, 34] shows that in MEIJE and SCCS, each process graph with r.e. sets of states and transitions can be finitely specified up to isomorphism. Each process graph that is effective in our sense clearly has r.e. sets of states and transitions, and can therefore be specified in MEIJE and SCCS. Since MEIJE and SCCS are clearly (recursively) enumerable, Theorem 2.6 tells us that the operational semantics for these languages is not effective. And in fact, it is easy to see that, due to the presence of unguarded recursion, these languages can specify process graphs with infinite branching. Boudol [14] points out that it is not even decidable whether a state has an outgoing transition.

3. Ponse [30] shows that in the calculus μCRL each effective process graph can be specified up to isomorphism. Here the twist is that the language μCRL, although enumerable, is not *recursively* enumerable. This makes that, even though for each individual μCRL program one can effectively compute the *root* and the Gödel number of the *next* function of the associated process graph, the operational semantics for the language as a whole is not effective with respect to any enumeration.

3 STRUCTURAL OPERATIONAL SEMANTICS

Plotkin [28, 29], advocates a simple method for giving operational semantics to programming languages. The method, which is often referred to as *SOS* (for *Structural Operational Semantics*), is based on the notion of transition systems. The states of the transition systems are elements of some formal language that may extend the language for which one wants to give an operational semantics. The main idea of the method is to define the transitions between states by a set of conditional rules over the syntax of the language, using structural induction. Because of its power and simplicity, the SOS approach has been highly successful and has become the standard way to equip programming languages with an operational semantics.

In this section we will recall some basic definitions and results from the theory of SOS.

3.1 SOS Calculi and Their Operational Semantics

DEFINITION 3.1 [Signatures and terms] To start with, we assume the presence of two disjoint countably infinite sets: a set $\mathcal{V}$ of *variables* with typical elements $x, y, \ldots$, and a set $\mathcal{N}$ of *names*. A *signature element* is a pair (f, n) consisting of a *function symbol* $f \in \mathcal{N}$ and an *arity* $n \in \mathsf{N}$. In a signature element $(c, 0)$, the c is often referred to as a *constant symbol*. A *signature* is a set of signature elements, *i.e.*, a subset of $\mathcal{N} \times \mathsf{N}$. The set of *terms* over a signature Σ is the smallest set $\mathbb{T}(\Sigma)$ with:

- $\mathcal{V} \subseteq \mathbb{T}(\Sigma)$,
- $(f, n) \in \Sigma$, $n \geq 0$, $t_1, \ldots, t_n \in \mathbb{T}(\Sigma)$ implies $f(t_1, \ldots, t_n) \in \mathbb{T}(\Sigma)$.

A term $c()$ is often abbreviated as c. $\mathrm{T}(\Sigma)$ is the set of *closed* terms over Σ, *i.e.*, terms in $\mathbb{T}(\Sigma)$ that do not contain variables. With $var(t)$ the set of variables occurring in t is denoted. For a term t, $|t|$ denotes the *size* of t, *i.e.*, the number of variables, and constant and function symbols occurring in t. A *substitution* ζ is a mapping from $\mathcal{V}$ to $\mathbb{T}(\Sigma)$. With $t[\zeta]$, we denote the result of the simultaneous substitution, for all x, of x by $\zeta(x)$:

- $x[\zeta] = \zeta(x)$,
- $f(t_1, \ldots, t_n)[\zeta] = f(t_1[\zeta], \ldots, t_n[\zeta])$.

The expression $t[t_1/x_1, \ldots, t_n/x_n]$ denotes the term obtained from t by simultaneous substitution of t_1 for x_1, t_2 for x_2, etc.

DEFINITION 3.2 [Contexts] Let Σ be a signature. A *context of n holes* C over Σ is a term in $\mathbb{T}(\Sigma)$ in which n variables occur, each variable only once. If $t_1, \ldots, t_n$ are terms over Σ, then $C[t_1, \ldots, t_n]$ denotes the term obtained by substituting t_1 for the first variable occurring in C, t_2 for the second variable, etc. Thus, if $x_1, \ldots, x_n$ are all different variables, $C[x_1, \ldots, x_n]$, denotes a context of n holes in which x_i is the i-th variable that occurs. A context is *trivial* if it consists of a single variable only.

Let Σ be a signature. An equivalence $\equiv$ on $T(\Sigma)$ is *preserved under contexts*, and it is a *congruence*, if for all contexts $C[x]$, $t \equiv t' \Rightarrow C[t] \equiv C[t']$.

LEMMA 3.3 *Let Σ be a signature and $\equiv$ an equivalence over $T(\Sigma)$. Then $\equiv$ is a congruence iff for all $(f, n) \in \Sigma$: $t_1 \equiv u_1 \ \wedge \ \cdots \ \wedge \ t_n \equiv u_n \ \Rightarrow \ f(t_1, \ldots, t_n) \equiv f(u_1, \ldots, u_n)$.*

DEFINITION 3.4 [Calculi] Let A be a given set of *labels* and let Σ be a signature. The set $Tr(\Sigma, A)$ of *transitions* consists of all expressions of the form $t \xrightarrow{a} t'$ with $t, t' \in \mathbb{T}(\Sigma)$ and $a \in A$. The symbols $\phi, \psi, \ldots$ will be used to range over transitions. The set $Cf(\Sigma, A)$ of *inference rules* or *conditional formulas* over Σ and A consists of all expressions $\dfrac{\psi_1, \ldots, \psi_n}{\psi}$, where $\psi_1, \ldots, \psi_n, \psi$ in $Tr(\Sigma, A)$. The transitions ψ_i are called the *antecedents* and ψ is called the *conclusion* of the rule. If no confusion can arise, a rule $\frac{}{\psi}$ is also written ψ. The notions "substitution" and "closed" extend to transitions and rules in the obvious way.
A *transition system specification* or *calculus* is a triple $P = (\Sigma, A, R)$ with Σ a signature, A a set of labels and $R \subseteq Cf(\Sigma, A)$ a set of rules. If P and P' are two calculi, then $P \cup P'$ is obtained by taking the pairwise union of the signatures and rules.

DEFINITION 3.5 [Proofs] Let $P = (\Sigma, A, R)$ be a calculus. A *proof* of a transition ψ from P is a finite tree whose edges are ordered and whose vertices are labeled by transitions in $Tr(\Sigma, A)$, such that:

- the root is labeled with ψ,
- if ϕ is the label of some vertex and $\phi_1, \ldots, \phi_n$ are the labels of the children of this vertex, then there is a rule $\frac{\chi_1, \ldots, \chi_n}{\chi} \in R$ and a substitution ζ such that $\phi_i = \chi_i[\zeta]$ and $\phi = \chi[\zeta]$.

If a proof tree for ψ exists, then ψ is *provable* from P, notation $P \vdash \psi$.

DEFINITION 3.6 [Operational semantics] Let $P = (\Sigma, A, R)$ be a calculus, The LTS $lts(P)$ is defined as $(T(\Sigma), \longrightarrow)$ where $(t, a, t') \in \longrightarrow$ iff $P \vdash t \xrightarrow{a} t'$. The operational semantics $\mathcal{O}_P$ is the mapping that associates to a closed term $t \in \mathrm{T}(\Sigma)$ the process graph $graph(t, lts(P))$.

The last definition in this subsection recalls the notion of a Σ-algebra.

DEFINITION 3.7 Let Σ be a signature. A Σ-*algebra* $\mathcal{A}$ consists of a set $D_{\mathcal{A}}$, the *domain* of $\mathcal{A}$, and a mapping that associates to each signature element $(f, n) \in \Sigma$ an n-ary operation $f_{\mathcal{A}}$ on $D_{\mathcal{A}}$. A *valuation* in a Σ-algebra $\mathcal{A}$ is a function ξ that takes every variable x into an element of $D_{\mathcal{A}}$. The ξ-*evaluation* $[\![\cdot]\!]^{\xi}_{\mathcal{A}} : \mathbb{T}(\Sigma) \to D_{\mathcal{A}}$ is defined inductively by

$$[\![x]\!]^{\xi}_{\mathcal{A}} \quad \triangleq \quad \xi(x),$$

$$[\![f(t_1, \ldots, t_n)]\!]^{\xi}_{\mathcal{A}} \quad \triangleq \quad f_{\mathcal{A}}([\![t_1]\!]^{\xi}_{\mathcal{A}}, \ldots, [\![t_n]\!]^{\xi}_{\mathcal{A}}).$$

The result of a ξ-evaluation of a term t depends only on the value assigned by ξ to the variables occurring in t. In particular, if t is a closed term, then $[\![t]\!]^{\xi}_{\mathcal{A}}$ does not depend on ξ at all. Thus we can write simply $[\![t]\!]_{\mathcal{A}}$ in such a situation.

A *congruence* on $\mathcal{A}$ is an equivalence relation $\equiv$ on $D_{\mathcal{A}}$ with the property that for all $(f, n) \in \Sigma$,

$$d_1 \equiv d_1' \wedge \cdots \wedge d_n \equiv d_n' \quad \Rightarrow \quad f_{\mathcal{A}}(d_1, \ldots, d_n) \equiv f_{\mathcal{A}}(d_1', \ldots, d_n').$$

For $\mathcal{A}$ a Σ-algebra and $\equiv$ a congruence on $\mathcal{A}$, the Σ-algebra $\mathcal{A}/\equiv$ is defined by

$$D_{\mathcal{A}/\equiv} \quad \triangleq \quad \{d/_{\equiv} \mid d \in D_{\mathcal{A}}\},$$

$$f_{\mathcal{A}/\equiv}(d_1/_{\equiv}, \ldots, d_n/_{\equiv}) \quad \triangleq \quad (f_{\mathcal{A}}(d_1, \ldots, d_n))/_{\equiv},$$

where, of course, $e/_{\equiv} = \{e' \mid e' \equiv e\}$. Due to the congruence property this definition is independent of the choice of the representing $d_i \in d_i/_{\equiv}$.

3.2 The Calculus PC

As a running example in this paper, we will now present the calculus PC (for Process Calculus).

We assume the presence of a countable set A of *actions*, ranged over by $a, b, \ldots$, and of a countable set $\mathcal{X}$ of *process names*, ranged over by $X, Y, \ldots$. The set of *process terms*, which has typical elements $p, q, \ldots$, is defined via the signature Σ_{PC} displayed in Table 1.

Infix notation will be used for the binary function symbols, and we write $a \cdot p$ instead of $a \cdot (p)$. To avoid parentheses, it will be assumes that prefixing has most binding power, followed by product, which in turn is followed by free merge, which is followed by alternative composition (which has the weakest binding power). In the case of several sum, merge or product operations we will mostly omit brackets since semantically these operations are associative. (Readers who insist on complete parsing information may assume that missing

$\mathbf{0}$	0	inaction
a	1	prefixing; for each $a \in A$
$+$	2	alternative composition, sum
$\|$	2	parallel composition, (free) merge
$\times$	2	synchronous composition, product
ρ_r	1	renaming; for each $r \subseteq A \times A$
X	0	process names; for each $X \in \mathcal{X}$

TABLE 1. The signature of PC.

brackets associate to the right.) For a finite index set $I = \{i_1, \ldots, i_n\}$ and process terms $p_{i_1}, \ldots, p_{i_n}$, $\sum_{i \in I} p_i$ abbreviates $p_{i_1} + \ldots + p_{i_n}$. By convention $\sum_{i \in \emptyset}$ stands for $\mathbf{0}$. Trailing $\mathbf{0}$'s will often be dropped.

The constant $\mathbf{0}$ denotes *inaction*, a process that cannot do anything at all. The process $a \cdot p$ first performs an a-action and then behaves like p. Process $p + q$ will behave either like p or like q. It is not specified whether the choice between p and q is made by the process itself or by the environment. With $p \| q$, we denote the parallel composition of p and q without any synchronization between the p and q. The product $p \times q$ denotes the parallel composition of p and q in which *all* actions have to synchronize. The operation ρ_r is a slight generalization of the renaming/relabeling operations in CCS, CSP, MEIJE and ACP. Process $\rho_r(p)$ behaves just like process p, except that if p has the possibility of doing an a, $\rho_r(p)$ can do any action b that is related to a via r. The recursive definitions of the process names are given by a *declaration* function $E : \mathcal{X} \to \mathrm{T}(\Sigma_{PC})$. The process expressions $E(X)$ may contain only guarded occurrences of process names. An occurrence of a process name is *guarded* if it occurs in a subexpression $a \cdot p$. The condition of guardedness is standard in process theory and excludes recursive declarations like $E(X) = X$ that give no clue about the specified behavior. Often we will write $X \Leftarrow t$ as abbreviation for $E(X) = t$.

Some references for those readers who are familiar with other work on process theory. The constant $\mathbf{0}$ also occurs in CCS and MEIJE, and plays the same role as δ in ACP. The $+$ is the same as in CCS and ACP. The $\|$ operator occurs in ACP, CCS and TCSP, and the $\times$ operator is taken from TCSP. The ρ_r operator can be viewed as a *generalized state operator* in the sense of Baeten and Bergstra [5] if one assumes a state space that contains only a single element. It is also possible to view this operator as a special case of the *action refinement* operator as studied by Goltz and Van Glabbeek [17]: ρ_r refines an action a into the nondeterministic sum of the actions in $\{b \mid r(a, b)\}$.

The inference rules of PC are presented in Table 2. In the table a and b range over A, unless further restrictions are made. Further r ranges over $A \times A$, and variables x, x', y and

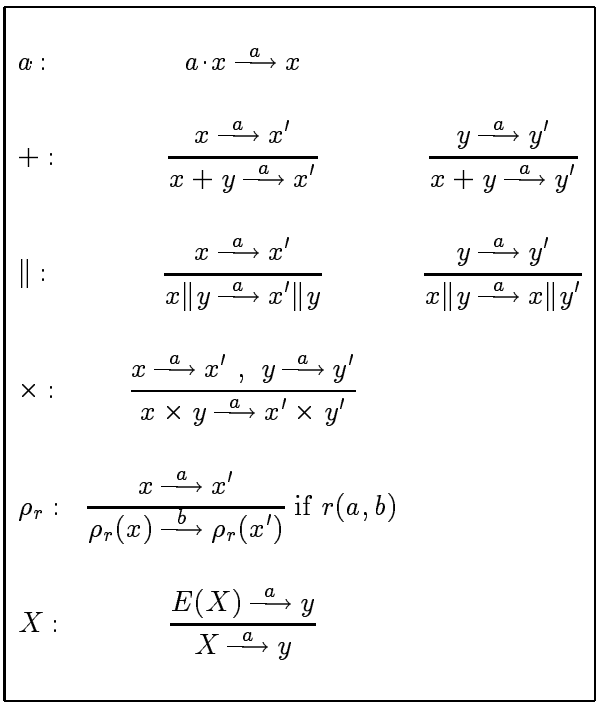

TABLE 2. The inference rules for PC.

y' are fixed and all different.

3.3 Power to Simulate 2-Counter Machines

The first (and weakest) form of universality that we consider is that a process calculus has the expressive power of 2-counter machines (or, equivalently, Turing machines) in the sense that, for each n, we can exhibit a term $U2CM_n$ whose process graph simulates in lock step a universal 2-counter machine on input n.

Calculi like CCS, CSP, ACP, and MEIJE are all universally expressive in this sense. Actually, trying to code a 2-counter or Turing machine in each of these languages is a nice way to get familiar with them. Via a rather tricky encoding, we prove below that also PC has the power of 2-counter machines.[1]

THEOREM 3.8 *PC has the expressive power of 2-counter machines.*

PROOF Suppose that a universal 2-counter machine has code of the form

l_1: `if I=0 goto` l_5
l_2: `inc I`
l_3: `dec J`
l_4: `goto` l_7
$\vdots$

[1] In Section 5, it will be shown how many operations can be defined as derived opertions of PC. Using derived operations like sequential composition, much simpler encodings can be obtained.

l_k: `halt`

The finite control part of this machine can be modeled by the PC expression *Control*, defined recursively by:

$$\begin{aligned}
Control &\Leftarrow X_1 \\
X_1 &\Leftarrow zero_I \cdot X_5 \ + \ non_zero_I \cdot X_2 \\
X_2 &\Leftarrow inc_I \cdot X_3 \\
X_3 &\Leftarrow dec_J \cdot X_4 \\
X_4 &\Leftarrow skip \cdot X_7 \\
&\vdots \\
X_k &\Leftarrow halt \cdot \mathbf{0}
\end{aligned}$$

The next step in the construction of a universal 2-counter machine is the following specification of a counter:

$$\begin{aligned}
C &\Leftarrow inc \cdot \rho_r((Syn \| Full) \times C) \ + \ \overline{dec} \cdot C \ + \ zero \cdot C \ + \ \overline{non_zero} \cdot C \\
Syn &\Leftarrow inc \cdot Syn \ + \ dec \cdot Syn \ + \ non_zero \cdot Syn \\
Full &\Leftarrow \overline{dec} \cdot Empty \ + \ \overline{non_zero} \cdot Full \\
Empty &\Leftarrow zero \cdot Empty
\end{aligned}$$

where

$$\begin{aligned}
r \ \triangleq \ \{&(inc, inc), (dec, dec), (\overline{dec}, dec)(zero, zero), (non_zero, non_zero), \\
&(zero, \overline{dec}), (zero, \overline{non_zero}), (\overline{non_zero}, non_zero)\}
\end{aligned}$$

Using the above recursive definitions, we can define, for each n, a PC expression representing a counter with value n:

$$\begin{aligned}
Counter_0 &\triangleq C, \\
Counter_{n+1} &\triangleq \rho_r((Syn \| Full) \times Counter_n).
\end{aligned}$$

Now the 2-counter machine with input n can be obtained by glueing together the finite control with 2 counters:

$$U2CM_n \ \triangleq \ Control \times (\rho_i(Counter_n) \| \rho_j(Counter_0)),$$

where

$$\begin{aligned}
i &\triangleq \{(inc, inc_I), (dec, dec_I), (\overline{dec}, dec_I), (zero, zero_I), (non_zero, non_zero_I)\} \\
j &\triangleq \{(inc, inc_J), (dec, dec_J), (\overline{dec}, dec_J), (zero, zero_J), (non_zero, non_zero_J)\}
\end{aligned}$$

■

3.4 Bisimulation Equivalence

A serious problem with the isomorphism relation $\simeq$ on process graphs is that it is not a congruence for calculi like CCS and PC. For instance, $\mathbf{0} \simeq \mathbf{0}\|\mathbf{0}$ but $a\cdot\mathbf{0}+a\cdot\mathbf{0} \not\simeq a\cdot\mathbf{0}+a\cdot(\mathbf{0}\|\mathbf{0})$. Thus it is not allowed to replace a subexpression by an $\simeq$-equivalent expression if one likes to preserve $\simeq$.

In order to remedy this problem, we will introduce the notion of *bisimulation equivalence*. Bisimulation equivalence is somewhat coarser than isomorphism and is a congruence with respect to all the constructs in the language. Actually, most researchers who use bisimulations motivate them in a different way. In [25, 8], for instance, it is at least suggested that two process terms are bisimilar iff they cannot be distinguished by an observer. We consider the arguments for bisimulation as a testing equivalence (see [1]) not really convincing and prefer to motivate this important notion in a different way. The following definition is essentially due to Park [26].

DEFINITION 3.9 [Bisimulation] Let $g_i = (r_i, S_i, \longrightarrow_i)$ $(i = 1, 2)$ be process graphs. A relation $R \subseteq S_1 \times S_2$ is a *(strong) bisimulation* between g_1 and g_2 if it satisfies:

1. $r_1 R r_2$;
2. if sRt and $s \xrightarrow{a}_1 s'$, then there exists a $t' \in S_2$ with $t \xrightarrow{a}_2 t'$ and $s'Rt'$;
3. if sRt and $t \xrightarrow{a}_2 t'$, then there exists an $s' \in S_1$ with $s \xrightarrow{a}_1 s'$ and $s'Rt'$.

Graphs g_1 and g_2 are *bisimilar*, notation $g_1 \leftrightarroweq g_2$, if there exists a bisimulation between them. Note that bisimilarity is an equivalence relation.

Two states s and s' of a LTS $\mathcal{A}$ are bisimilar iff $graph(s, \mathcal{A})$ and $graph(s', \mathcal{A})$ are bisimilar.

Two closed terms t, t' are *bisimilar* with respect to a calculus P, notation $P : t \leftrightarroweq t'$, if $\mathcal{O}_P(t) \leftrightarroweq \mathcal{O}_P(t')$.

Since any isomorphism between two process graphs is also a bisimulation, it follows that $\simeq$ is contained in $\leftrightarroweq$. It turns out that, just like isomorphism, bisimulation equivalence is not a congruence relative to all transition system specifications. For instance, if one adds to the calculus PC a rule $x+x \xrightarrow{b} \mathbf{0}$, then one can show that $\mathbf{0} \leftrightarroweq \mathbf{0}\|\mathbf{0}$ but $\mathbf{0}+\mathbf{0} \not\leftrightarroweq \mathbf{0}+\mathbf{0}\|\mathbf{0}$. In [19] the question under which conditions bisimulation is a congruence is considered in great depth. It turns out that if the rules in a calculus fit the very general *tyft/tyxt* format, bisimulation is a congruence. We will not discuss the *tyft/tyxt* format here, but instead present a more restricted format, essentially due to De Simone [33, 34], which is sufficiently general for our purposes. The reader may check that the transition system specification for the language PC fits this format.

DEFINITION 3.10 [De Simone's format] Let $\{x_i \mid i \in \mathsf{N}\}$ and $\{y_j \mid j \in \mathsf{N}\}$ be two fixed sets of variables in $\mathcal{V}$ with all x_i and y_j different. Let Σ be a signature and let A be a set of labels.

A rule in $Cf(\Sigma, A)$ is a *De Simone* rule if it takes the form:

$$\frac{x_i \xrightarrow{a_i} y_i \quad (i \in I)}{f(x_1, \ldots, x_n) \xrightarrow{a} t}$$

where

- $(f, n) \in \Sigma$;
- $I \subseteq \{1, \ldots, n\}$;
- if, for $1 \leq i \leq n$, $z_i = y_i$ if $i \in I$ and $z_i = x_i$ otherwise, then $t \in \mathbb{T}(\Sigma)$ is a context with variables in $\{z_1, \ldots, z_n\}$ (so each variable occurs at most once).

In the above rule, (f, n) is the *type*, a the *action*, t the *target*, and the tuple $\langle l_1, \ldots, l_n \rangle$ with $l_i = a_i$ if $i \in I$ and $l_i = *$ otherwise, is the *trigger*. If $i \in I$, then the i-th position is *active* in the rule; otherwise it is *passive*. Each rule is characterized uniquely by its type, action, target and trigger. If r is a rule, these ingredients will be referred to as $type(r)$, $action(r)$, $target(r)$ and $trigger(r)$.

A calculus (Σ, A, R) is a *De Simone system* if Σ can be partitioned into Σ_1 and Σ_2, and R can be partitioned into R_1 and R_2 in such a way that:

- all the rules in R_1 are De Simone rules with a type in Σ_1;
- there exists a set $\mathcal{X} \subseteq \mathcal{N}$ and a mapping $E : \mathcal{X} \to \mathbb{T}(\Sigma)$ such that:

$$\Sigma_2 = \{(X, 0) \mid X \in \mathcal{X}\} \text{ and } R_2 = \{\frac{E(X) \xrightarrow{a} y_0}{X \xrightarrow{a} y_0} \mid X \in \mathcal{X} \text{ and } a \in A\}.$$

Elements of $\mathcal{X}$ are referred to as *process names* and E is called the *declaration mapping*. If a calculus P is a De Simone system, then both the set of process names and the declaration mapping are uniquely determined and will be referred to as $\mathcal{X}_P$ and E_P.

THEOREM 3.11 *Let P be a De Simone system. Then bisimulation equivalence with respect to P is a congruence on the signature of P.*

PROOF Standard. This theorem was first proved in [33] in a slightly different setting. The theorem is in fact a corollary of some of the other results in this paper (see remark at the end of Section 5.1). ■

4 Power to Specify Graphs

In this section we will present what one could call bad news: in the case of 'strong' equivalences the expressiveness of SOS languages is, in many cases, even less than suggested by Theorem 2.6. For a rather large class of languages we can give an upper bound on the speed with which the fanout can grow. This upper bound implies that, for each of these languages, there exists a primitive recursive process graph that cannot be denoted up to (strong) trace equivalence.

4.1 Effective De Simone Systems

In this subsection, we will identify a class of De Simone systems that induce an effective operational semantics. This result is a useful, because if one is able to define an operational semantics for a programming language by means of a calculus in this class, then one knows that (at least in principle) it is possible to implement the language.

Bloom, Istrail and Meyer [12] introduce a particular format of transition system specifications, which they call *GSOS rule system*'s, and show that for any specification in this format the associated operational semantics is effective and has some other desirable properties as well. The authors argue that it is not possible to generalize the GSOS format in any obvious way without loosing one of these desirable properties. However, one of the clauses in the definition of the GSOS format is that the number of rules must be finite. We think that this clause is unnecessarily restrictive and hinders application of the nice theory developed for this format. Calculi like CCS, SCCS and MEIJE all have an infinite number of actions and an infinite number of rules. Consequently it is not possible to view them as GSOS rule systems, even if one restricts attention to subcalculi with guarded recursion. Below, we introduce the increasingly restrictive notions of *guarded*, *bounded* and *effective* De Simone systems. A bounded De Simone system associates to each term a finitely branching process graph. An effective De Simone system guarantees all the nice properties required in [12] and in particular that the induced operational semantics is effective. It will turn out that CCS-like calculi with guarded recursion can be viewed as effective De Simone systems. We claim that a similar restriction can replace the finiteness constraint in GSOS rule systems without any of the desired properties getting lost.

DEFINITION 4.1 [Guardedness] Let $P = (\Sigma, A, R)$ be a De Simone system, let (f, n) be a signature element of Σ, and let $1 \leq i \leq n$. Then (f, n) *tests* its i-th argument and the i-th argument is *awake* if there is a rule in R of type (f, n) in which the i-th position is active (*i.e.*, i occurs in the index set of the rule); otherwise the i-th position is *sleeping*. A term $t \in \overline{\Pi}(\Sigma)$ is *guarded* if all process names in t occur in subterms that are on a sleeping position. P is *guarded* if all terms in the image of E_P are guarded.

The only signature elements of PC with a sleeping position are the prefixing operations. Thus, the above notion of guardedness generalizes the definition of guardedness for the language PC. The notion of an operator testing an argument is due to Bloom [11]. However, the use of this notion in a definition of guardedness is new in the present paper.

DEFINITION 4.2 [Boundedness] A guarded De Simone system is *bounded* if for each type that is not a process name and for each trigger, the corresponding set of rules is finite.

The inference rules for the operations of CCS, SCCS and MEIJE, which are all in De Simone's format, have the property that for a given type and a given trigger there is only a single rule. If the action alphabet is infinite, then the De Simone system for PC is not bounded, due to the generalized renaming operator. In the case of PC it is easy to see that this unboundedness leads to infinite branching. Thus, if one prefers to have finite branching then one has to restrict attention to a subset of PC with renaming operations that relate each action to at most finitely many other actions.

THEOREM 4.3 *Let P be a bounded De Simone system. Then the operational semantics $\mathcal{O}_P$ maps each term to a finitely branching process graph.*

PROOF Routine and omitted. ■

DEFINITION 4.4 A bounded De Simone system $P = (\Sigma, A, R)$ is *effective*, relative to enumerations $(f_1, n_1), (f_2, n_2), \ldots$ and $a_1, a_2, \ldots$ of Σ and A, respectively, if:

- the set of process names is recursive;
- for each type that is not a process name and for each argument, it is decidable whether this argument is tested or not;
- for a given type that is not a process name and a given trigger, the cardinality of the corresponding (finite) set of rules as well as the set itself are recursive;
- the function E_P is recursive.

If the action alphabet is infinite, then PC is not effective. Effective versions of PC can be obtained by allowing only renaming relations r which relate an action to at most finitely many other actions, in such a way that for each a_i the canonical index of $\{j \mid r(a_i, a_j)\}$ is recursive. Some additional restrictions will be needed to make the language recursively enumerable or recursive.

THEOREM 4.5 *Each effective De Simone system induces an effective operational semantics.*

PROOF Routine and omitted. ■

4.2 The Expressiveness of CCS-like Languages

In his Ph.D. thesis, Bloom [10] shows that no GSOS rule system can denote all effective process graphs up to strong bisimulation. This result is an immediate corollary of Theorem 2.6 of this paper and the basic result about the GSOS format of [12], which says that this format induces an effective operational semantics. A nice aspect of Bloom's proof however is that the counterexample which he produces (for any GSOS-language an effective graph that cannot be denoted up to bisimulation equivalence) is quite simple and provides additional insight in the expressive power of GSOS languages. Bloom's proof uses two lemma's. The first lemma provides, for a given set of rules, an upper bound on the *fanout* of a term (*i.e.*, the number of outgoing transitions) that only depends on the size of that term. The second lemma provides, given a set of rules and a transition $p \xrightarrow{a} q$, an upper bound on the size of successor q in terms of the size of p. The combination of the two lemma's implies that in any GSOS-specifiable process graph the rate at which the fanout can grow is bounded. Using this observation it is easy to construct a counterexample.

Below, we will adapt Bloom's idea to the setting of this paper. In the case of a De Simone system it is not possible to give upper bounds on the fanout and the size of successor states of p in terms of p. If p is a process name then, depending on the size of the recursive definition of this process name, fanout and successors of p can be arbitrarily big. Therefore, we will give upper bounds on the fanout and the size of successors in terms of the size of p and the supremum over all process names X that occur unguarded in p of the size of the recursive definition for X.

The different treatment of recursion and the different finiteness constraints make any comparison nontrivial but, due to the fact that De Simone rules are more restricted than GSOS-rules, it appears that the upper bounds which we derive are smaller than those of Bloom [10].

A closer investigation of these bounds will be interesting because it might lead to a proof that certain process graphs are GSOS definable but not definable using De Simone systems.

We start off by defining, for each De Simone system, some parameters which will determine the possible growth rate in the process graphs.

DEFINITION 4.6 Let $P = (\Sigma, A, R)$ be a De Simone system.

- $\alpha_P \in \mathsf{N} \cup \{\infty\}$ is the supremum of 1 and, for each rule in R, the number of function symbols occurring in the target.
- $\beta_P \in \mathsf{N} \cup \{\infty\}$ is the supremum over all process names X of the size of $E_P(X)$.
- γ_P is the supremum over all types and triggers of the number of rules in R with that type and that trigger.

We write α, β and γ if P is clear from the context.

It is probably useful to illustrate this definition with some examples. In the De Simone system for PC, the α-parameter has value 1. In fact, and this is interesting to note, in most major process calculi proposed in the literature, the α-parameter is 1. One exception is the *desynchronising* operator Δ, present in an earlier version of SCCS and needed by De Simone [34] in order to show equivalence of SCCS and MEIJE. The Δ operator has an α parameter of 2:

$$\frac{x \xrightarrow{a} x'}{\Delta x \xrightarrow{a} \delta \Delta x'} \qquad \delta x \xrightarrow{a} \delta x \qquad \frac{x \xrightarrow{a} x'}{\delta x \xrightarrow{a} x'}$$

Another example of an operator with an α-parameter of 2 is the is the p `watching` S construct from synchronous programming language Esterel [9].

It is possible to have an infinite number of recursive definitions in a De Simone system and still have the β-parameter finite for each expression. For instance, consider the following infinitary PC definition of a counter:

$$C_0 = zero \cdot C_0 + up \cdot C_1$$

$$C_{n+1} = down \cdot C_n + up \cdot C_{n+2}.$$

One can easily check that the β-parameter of this system is 5.

Due to the presence of the relational renaming operations, the γ-parameter for PC is $|A|$: if r is the universal relation, then for any trigger $\langle a \rangle$, ρ_r has $|A|$ rules. Process calculi like CCS, SCCS and MEIJE all have a γ of 1.

Below, we will show that for any guarded De Simone system P with α, β and γ finite, there are strong bounds on the speed with which branching can grow.

LEMMA 4.7 *Let P be a guarded De Simone system with a transition $p \xrightarrow{a} q$. Suppose that α_P and β_P are finite. Then:*

$$|q| \leq \begin{cases} \alpha \cdot |p| & \text{if } p \text{ is guarded} \\ \alpha \cdot \beta \cdot |p| & \text{otherwise} \end{cases}$$

PROOF First, consider the case that p is guarded. By induction on the size of p, we prove that $|q| \leq \alpha \cdot |p|$.

Consider a proof of $p \xrightarrow{a} q$. Since p is guarded, it is not a process name. So the last inference rule used in the proof must be a De Simone rule. Let this rule be

$$\frac{x_i \xrightarrow{a_i} y_i \quad (i \in I)}{f(x_1, \ldots, x_n) \xrightarrow{a} t}$$

and let σ be the substitution by which this rule is instantiated. Then for all $i \in I$ the term $\sigma(x_i)$ is guarded and a proper subterm of p. Therefore the induction hypothesis can be used to conclude that for all $i \in I$, $|\sigma(y_i)| \leq \alpha \cdot |\sigma(x_i)|$. Because the rule is in De Simone's format, it follows that for all $1 \leq i \leq n$, t contains at most one occurrence of either x_i or y_i. This observation can be used to derive:

$$\begin{aligned} |q| &= |\sigma(t)| \leq \alpha + \alpha \cdot |\sigma(x_1)| + \ldots + \alpha \cdot |\sigma(x_n)| = \\ &= \alpha \cdot (1 + |\sigma(x_1)| + \ldots + |\sigma(x_n)|) = \alpha \cdot |\sigma(f(x_1, \ldots, x_n))| = \alpha \cdot |p|. \end{aligned}$$

This completes the proof for the case p is guarded.

Next, we prove, by induction on the size of p, that $|q| \leq \alpha \cdot \beta \cdot |p|$ if p is not guarded.

If p is of the form X, with X a process name, then $E_P(X) \xrightarrow{a} q$. But since $E_P(X)$ is guarded, the statement proved in the above can be used to derive:

$$|q| \leq \alpha \cdot |E_P(X)| \leq \alpha \cdot \beta = \alpha \cdot \beta \cdot |p|.$$

So assume that p is not a process name. Consider a proof of $p \xrightarrow{a} q$. The last inference rule used in the proof must be a De Simone rule. By an inductive argument which is similar to the one used for the guarded case it follows that $|q| \leq \alpha \cdot \beta \cdot |p|$. ■

LEMMA 4.8 *Let P be a guarded De Simone system and let p be a closed expression over the signature of P. Suppose that β_P and γ_P are finite. Then:*

$$fan(p) \leq \begin{cases} \gamma^{|p|} \cdot 2^{|p|-1} & \text{if } p \text{ is guarded} \\ \gamma^{\beta \cdot |p|} \cdot 2^{\beta \cdot |p|-1} & \text{otherwise} \end{cases}$$

PROOF First, consider the case that p is guarded. By induction on the size of p, we prove $fan(p) \leq \gamma^{|p|} \cdot 2^{|p|-1}$.

Let $p = f(p_1, \ldots, p_n)$. Then (f, n) is not a process name, and for each argument i that is tested by (f, n) the term p_i is guarded. Consider the collection $\mathcal{I}$ of pairs $(r, ((u_1, q_1), \ldots, (u_n, q_n)))$ satisfying

- r is a rule with type (f, n) and trigger $\langle u_1, \ldots, u_n \rangle$, and
- for each i either $u_i = q_i = *$ or $p_i \xrightarrow{u_i} q_i$.

It is not hard to see that $\mathcal{I}$ has at most $\gamma \cdot \prod_{i \text{ tested}} (fan(p_i) + 1)$ elements. Since there is a straightforward surjective mapping from $\mathcal{I}$ to the transitions of p, it follows that

$$fan(p) \leq \gamma \cdot \prod_{i \text{ tested}} (fan(p_i) + 1).$$

By induction hypothesis we obtain, for each i that is tested, $fan(p_i) \leq \gamma^{|p_i|} \cdot 2^{|p_i|-1}$. Thus we can derive:

$$\begin{aligned} fan(p) &\leq \gamma \cdot \prod_{i \text{ tested}} (fan(p_i) + 1) \leq \\ &\leq \gamma \cdot \prod_{i \text{ tested}} (\gamma^{|p_i|} \cdot 2^{|p_i|-1} + 1) \leq \\ &\leq \gamma \cdot \prod_{i=1}^{n} (\gamma^{|p_i|} \cdot 2^{|p_i|-1} + 1) \leq \\ &\leq \gamma \cdot \prod_{i=1}^{n} (\gamma^{|p_i|} \cdot 2^{|p_i|}) \leq \\ &\leq \gamma^{|p|} \cdot 2^{|p|-1}. \end{aligned}$$

This completes the proof for the case p is guarded.

Next, we prove, by induction on the size of p, that $fan(p) \leq \gamma^{\beta \cdot |p|} \cdot 2^{\beta \cdot |p|-1}$ if p is not guarded. If p is of the form X for some process name X, then $E_P(X)$ is guarded and of size less or equal than β. Hence:

$$fan(p) = fan(E_P(X)) \leq \gamma^{\beta} \cdot 2^{\beta-1} = \gamma^{\beta \cdot |p|} \cdot 2^{\beta \cdot |p|-1}.$$

So assume that p is of the form $f(p_1, \ldots, p_n)$ with (f, n) not a process name. By an inductive argument very similar to that used for the guarded case one can show $fan(p) \leq \gamma^{\beta \cdot |p|} \cdot 2^{\beta \cdot |p|-1}$. ■

THEOREM 4.9 *Let A be a countably infinite set of actions. Then there exists a primitive recursive process graph g over A that cannot be denoted modulo trace equivalence by any guarded De Simone system over alphabet A with α_P, β_P and γ_P finite.*

PROOF Without loss of generality assume $A = \mathsf{N}$. Let $P = (\Sigma, A, R)$ be a guarded De Simone system with α_P, β_P and γ_P finite. Suppose

$$p = p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} p_n$$

is a sequence of transitions starting in p. Then, by Lemma 4.7, we have for all $i < n$: $|p_{i+1}| \leq \alpha \cdot \beta \cdot |p_i|$. Thus, for all i, $|p_i| \leq \alpha^i \cdot \beta^i \cdot |p|$. Combining this result with Lemma 4.8 yields:

$$fan(p_i) \leq \gamma^{\alpha^i \cdot \beta^{i+1} \cdot |p|} \cdot 2^{\alpha^i \cdot \beta^{i+1} \cdot |p|-1}.$$

Thus, if $NT(n)$ is the number of different traces of length n in $\mathcal{O}_P(p)$, we have:

$$\begin{aligned} NT(n) &\leq \prod_{i=0}^{n-1} \gamma^{\alpha^i \cdot \beta^{i+1} \cdot |p|} \cdot 2^{\alpha^i \cdot \beta^{i+1} \cdot |p|-1} \\ &< \gamma^{\alpha^{n-1} \cdot \beta^n \cdot n \cdot |p|} \cdot 2^{\alpha^{n-1} \cdot \beta^n \cdot n \cdot |p|} \end{aligned}$$

$$1 \xrightarrow{1} 0$$

$$\frac{x \xrightarrow{n} x'}{succ(x) \xrightarrow{n+1} 0}$$

$$triple(x) \xrightarrow{0} triple(succ(x))$$

$$\frac{x \xrightarrow{n} x'}{triple(x) \xrightarrow{m} 0} \text{ if } 0 < m < 2^{2^{2^n}}$$

TABLE 3. Rules for process graph with triple exponential growth rate.

Even though $NT(n)$ can grow fast, its growth rate is still double exponential.

Let *Triple* be the calculus with a signature consisting of two constant symbols 1 and 0, two unary function symbols *succ* and *ack*, and rules as given in Table 3. Now define g to be the process graph $\mathcal{O}_{Triple}(triple(1))$. It is easy to see that graph g is primitive recursive, and also that, for each n, it has $2^{2^{2^n}}$ different traces of length n. Thus, if n is chosen sufficiently large, then there is some trace of length n in graph g, that is not a trace of the graph of p: a routine exercise tells us that, for big enough n, $NT(n) < 2^{2^{2^n}}$. Thus it cannot be the case that $\mathcal{O}_P(p) \approx_T g$. ■

A corollary of the above result is that graph g can not be specified in the guarded, finitary versions of calculi like CCS, SCCS, MEIJE and ACP.

THEOREM 4.10 *The graph g can be specified in an effective version of PC.*

PROOF Take as the alphabet of actions the set N of natural numbers. Define the relations $Succ$ and $Triple$ by:

$$\begin{aligned} Succ &\triangleq \{(0,0)\} \cup \{(n, n+1) \mid n > 0\}, \\ Triple &\triangleq \{(0,0)\} \cup \{(n,m) \mid n, m > 0 \text{ and } m < 2^{2^{2^n}}\}. \end{aligned}$$

Let X be a process name with recursive definition $X \Leftarrow 0 \cdot \rho_{Succ}(X) + 1$. Then it is straightforward to check that the term $\rho_{Triple}(X)$ denotes g up to isomorphism. ■

5 POWER TO SPECIFY OPERATIONS ON GRAPHS

The third way in which a process calculus can be universal is that all operations in a given natural class can be defined in terms of the operations in the language modulo a given equivalence. The first result of this kind occurring in the literature is due to De Simone [33, 34], who shows that all operations that can be defined in a format similar to what we call De Simone's format in this paper, are definable in terms of both the languages MEIJE and SCCS up to (strong) bisimulation equivalence. Another result is due to Parrow [27], who shows that all *network* operators specifiable in a restricted De Simone format can be defined

up to weak bisimulation equivalence in terms of only two operators: *disjoint parallelism* and *linking*.

5.1 From Calculi to Operations on Graphs

Strictly speaking, the above phrasing of De Simone's result is not correct. What he shows in fact is that for any calculus in a particular format, and for any n-ary function symbol f from that calculus, there exists a MEIJE-SCCS context which is "FH-bisimilar" with the expression $f(x_1, \ldots, x_n)$. Clearly, there is a close connection between the notion of FH-bisimilarity and the equality of certain operations on process graphs modulo bisimulation. However, this is left implicit in De Simone's work. It is not even made clear how a calculus determines operations on process graphs.

A first contribution of this section is a precise definition of the transformation from a calculus to operations on graphs. Although the result is the same, the definition of the transformation that we present here is quite different from the definition in Baeten and Vaandrager [7]. Our definition, which in spirit is very close to De Simone's notion of FH-bisimulation, turns out to be useful for proving that an operation from one calculus is a derived operation from another calculus.

Technically, a key role is played by the notion of an *action transducer*: to each function symbol in a given calculus a (rooted) action transducer is associated, which in turn determines an operation on process graphs. Action transducers were introduced by Larsen and Xinxin [22] as a technical tool for proving certain compositionality results. An action transducer is an object that consumes actions provided by its internal processes, in return produces an action for an external observer, and may change as a result of this transduction. The definition of an action transducer below differs from the corresponding definition of a context system in [22], and captures explicitly the possibility that in a dynamic situation a context may now and then lose some of its holes. The idea to associate an action transducer to a calculus using the notion of a *linear proof* is also new in this paper.

DEFINITION 5.1 An *action transducer* over a set A of actions is a triple $T = (\mathcal{C}, h, \longrightarrow)$, where

- $\mathcal{C}$ is a countable set of *contexts*;
- h is a mapping from $\mathcal{C}$ to finite subsets of N, which associates *holes* to contexts;
- $\longrightarrow$ is a subset of $\mathcal{C} \times A \times Pow(\mathsf{N} \times A) \times \mathcal{C}$ with for each $(C, a, \eta, C') \in \longrightarrow$, $h(C') \subseteq h(C)$ and η a function with $domain(\eta) \subseteq h(C)$.

Elements of $\longrightarrow$ are called *transductions*, and we write $C \xrightarrow[\eta]{a} C'$ if $(C, a, \eta, C') \in \longrightarrow$.

A *rooted action transducer* or *operator graph* over A is a tuple $(C_0, \mathcal{C}, h, \rightarrow)$, where $(\mathcal{C}, h, \rightarrow)$ is an action transducer over A, $C_0 \in \mathcal{C}$ is the *root*, and each context in $\mathcal{C}$ is reachable via zero or more transductions from C_0. If T is an action transducer and C is a context of T, then $og(C, T)$ is the operator graph with root C and an underlying action transducer that is obtained by restricting T to the part that is reachable from C.

DEFINITION 5.2 [Graph domain] $\mathcal{G}(A)$ is the set of process graphs with states taken from N and transition labels from A. For $g \in \mathcal{G}(A)$, $[g]$ denotes the isomorphism class of g. $\widehat{\mathcal{G}}(A)$ is the set of isomorphism classes of $\mathcal{G}(A)$.

DEFINITION 5.3 [From action transducers to operations on graphs] Let $F = (C_0, \mathcal{C}, h, \longrightarrow)$ be an operator graph over A with $h(C_0) = \{i_1, \ldots, i_n\}$. To F an n-ary operator $op(F)$ on $\widehat{\mathcal{G}}(A)$ is associated as follows. Assume w.l.o.g. that $i_j < i_k$ for $1 \leq j < k \leq n$. Let, for $1 \leq j \leq n$, $g_j = (r_j, S_j, \longrightarrow_j) \in \mathcal{G}(A)$. Then $op(F)([g_1], \ldots, [g_n])$ is the isomorphism class in $\hat{\mathcal{G}}(A)$ of graphs that are isomorphic to the process graph $graph(r, (S, \longrightarrow))$ where

- $r = (C_0, r_1, \ldots, r_n)$;
- $S = \mathcal{C} \times S_1 \times \cdots \times S_n$;
- $(C, s_1, \ldots, s_n) \xrightarrow{a} (C', s'_1, \ldots, s'_n)$ iff there is an η such that $C \xrightarrow[\eta]{a} C'$ and for $1 \leq j \leq n$, $i_j \notin domain(\eta) \Rightarrow s_j = s'_j$ and $\forall b \in A : (i_j, b) \in \eta \Rightarrow s_j \xrightarrow{b}_j s'_j$.

LEMMA 5.4 *Let $F = (C_0, \mathcal{C}, h, \longrightarrow)$ be an operator graph, with $op(F)$ an n-ary operator on $\hat{\mathcal{G}}(A)$. Let for $1 \leq i \leq n$, $g_i, g'_i \in \mathcal{G}(A)$. Then*

$$\forall i : g_i \leftrightarrows g'_i \quad \Rightarrow \quad op(F)([g_1], \ldots, [g_n]) \leftrightarrows op(F)([g'_1], \ldots, [g'_n]).$$

PROOF Suppose that for all i, $g_i \leftrightarrows g'_i$. Let R_i be a bisimulation between g_i and g'_i. Define the relation R between states of $op(F)([g_1], \ldots, [g_n])$ and states of $op(F)([g'_1], \ldots, [g'_n])$ by

$$(C, s_1, \ldots, s_n) R (C, s'_1, \ldots, s'_n) \text{ iff } \forall i : s_i \leftrightarrows s'_i.$$

It is easy to check that R is a bisimulation, from which it follows that $op(F)([g_1], \ldots, [g_n]) \leftrightarrows op(F)([g'_1], \ldots, [g'_n])$. ■

We will now define how action transducers can be associated to De Simone calculi. The obvious choice for the contexts of the action transducer are the contexts of the De Simone calculi (open terms over the signature in which variables occur linearly). In an attempt to emphasize that an SOS calculus is essentially a logical theory, the transductions of the action transducer will be defined in terms of conditional formulas that are provable from the calculus.

DEFINITION 5.5 [Linear proofs] Let $P = (\Sigma, A, R)$ be a calculus. A *linear proof* from P of a conditional formula $\rho = \frac{\psi_1, \ldots, \psi_n}{\psi} \in Cf(\Sigma, A)$ is a finite tree whose edges are ordered and whose vertices are labeled by transitions in $Tr(\Sigma, A)$, such that:

- the root is labeled with ψ;
- there are distinct vertices $v_1, \ldots, v_n$ in the tree, which occur as leaves and are labeled with $\psi_1, \ldots, \psi_n$, respectively;

- if ϕ is the label of a node $v \notin \{v_1, \ldots, v_n\}$ and $\phi_1, \ldots, \phi_m$ are the labels of the children of v, then there is a rule $\frac{\chi_1, \ldots, \chi_m}{\chi} \in R$ and a substitution ζ such that $\phi_i = \chi_i[\zeta]$ and $\phi = \chi[\zeta]$.

Write $P \vdash_L \rho$ if a linear proof of ρ from P exists.

The term "linear" is used because of the apparent connection with the Linear Logic of Girard [16]. In a linear proof of a conditional formula, each hypothesis is used exactly once. This "resource consciousness" should be contrasted with proofs in non-linear conditional logics, in which an hypothesis may be used several times, or not at all. The notion of linear provability generalizes the proof notion of Definition 3.5 in the sense that for closed terms t, t',

$$P \vdash t \xrightarrow{a} t' \text{ iff } P \vdash_L \frac{}{t \xrightarrow{a} t'}.$$

The following lemma is easily proved by induction on the structure of linear proofs.

LEMMA 5.6 *Let P be a De Simone calculus with*

$$P \vdash_L \frac{x_i \xrightarrow{a_i} x_i \quad (i \in I)}{C \xrightarrow{a} C'},$$

where C is a context with variables from $\{x_i \mid i \in \mathsf{N}\}$. Then C' is a context, $\{x_i \mid i \in I\} \subseteq var(C)$, and $var(C') \subseteq var(C)$.

DEFINITION 5.7 [From calculi to transducers] To each De Simone system $P = (\Sigma, A, R)$, an action transducer $transducer(P) = (\mathcal{C}, h, \longrightarrow)$ is associated as follows.

- $\mathcal{C}$ consists of the contexts in $\mathbb{T}(\Sigma)$ with variables in $\{x_i \mid i \in \mathsf{N}\}$;
- h associates to each context the set of indices of its variables;
- Let $C, C' \in \mathcal{C}$, $a \in A$, and $\eta = \{(i, a_i) \mid i \in I\}$ a finite subset of $\mathsf{N} \times A$. Then

$$C \xrightarrow[\eta]{a} C' \quad \text{iff} \quad P \vdash_L \frac{x_i \xrightarrow{a_i} x_i \quad (i \in I)}{C \xrightarrow{a} C'}.$$

It follows using Lemma 5.6 that $transducer(P)$ is indeed a transducer.

The use of premisses $x_i \xrightarrow{a_i} x_i$ in the above definition may appear strange at first sight: after performing a transition an agent does not in general evolves into itself and therefore the hypotheses seem too strong. However, this turns out not too be the case: one can prove by straightforward induction on the structure of proofs that

$$P \vdash_L \frac{x_i \xrightarrow{a_i} x_i \quad (i \in I)}{C \xrightarrow{a} C'} \quad \text{iff} \quad P \vdash_L \frac{x_i \xrightarrow{a_i} y_i \quad (i \in I)}{C \xrightarrow{a} C'[y_i / x_i (i \in I)]}.$$

Thus, modulo syntactic details, the transductions in $transducer(P)$ are *exactly* the formulas that can be derived using a linear form of logical inference.

DEFINITION 5.8 [From SOS contexts to operators] Let $P = (\Sigma, A, R)$ be a De Simone system and let C be an n-ary context over Σ and $\{x_i \mid i \in \mathsf{N}\}$. The n-ary operator $\langle\!\langle C \rangle\!\rangle_P$ on $\widehat{\mathcal{G}}(A)$ is given by

$$\langle\!\langle C \rangle\!\rangle_P = op(og(C, transducer(P))).$$

For $h(C) = \{i_1, \ldots, i_n\}$ with $j < k \Rightarrow i_j < i_k$, and $\xi : var(C) \to \widehat{\mathcal{G}}(A)$, the process graph $\langle\!\langle C \rangle\!\rangle_P^{\xi}$ is defined by

$$\langle\!\langle C \rangle\!\rangle_P^{\xi} = \langle\!\langle C \rangle\!\rangle_P(\xi(x_{i_1}), \ldots, \xi(x_{i_n})).$$

The following two technical lemmas play a key role in the further developments of this section.

LEMMA 5.9 *Let P be a De Simone system and let t be a closed term over the signature of P. Then $\mathcal{O}_P(t) \simeq \langle\!\langle t \rangle\!\rangle_P()$.*

PROOF Straightforward. ■

LEMMA 5.10 *Let P be a De Simone system. Let $C, C_1, \ldots, C_n$ be contexts over the signature of P with $var(C) = \{x_1, \ldots, x_n\}$ and $var(C_i) \subseteq \{x_i \mid i \in \mathsf{N}\}$ such that $k \neq l \Rightarrow var(C_k) \cap var(C_l) = \emptyset$. Let ξ_i be mappings from $var(C_i)$ to $\widehat{\mathcal{G}}(A)$. Then*

$$\langle\!\langle C \rangle\!\rangle_P(\langle\!\langle C_1 \rangle\!\rangle_P^{\xi_1}, \ldots, \langle\!\langle C_n \rangle\!\rangle_P^{\xi_n}) \leftrightarroweq \langle\!\langle C[C_1/x_1, \ldots, C_n/x_n] \rangle\!\rangle_P^{\xi_1 \cup \cdots \cup \xi_n}.$$

DEFINITION 5.11 [From calculi to process algebras] Let $P = (\Sigma, A, R)$ be a De Simone system. The Σ-algebra $\mathcal{A}(P)$ has as domain $\widehat{\mathcal{G}}(A)$; each signature element (f, n) is mapped to the n-ary operation $f_{\mathcal{A}(P)} = \langle\!\langle f(x_1, \ldots, x_n) \rangle\!\rangle_P$.

For a given De Simone system P, the evaluation function $[\![\cdot]\!]_{\mathcal{A}(P)}$ maps each closed term to an isomorphism class of process graphs. An obvious question is how this compositional semantics relates to the operational semantics $\mathcal{O}_P$. It turns out that the two mappings are different if we consider them up to isomorphism. The counterexample is similar to the one used in Section 3.4 to show that isomorphism is not a congruence for PC:

$$[\![a + a]\!]_{\mathcal{A}(PC)} \not\simeq \mathcal{O}_{PC}(a + a).$$

However, as we will see, the two mapping are the same modulo bisimulation equivalence. Notice that, due to Lemma 5.4, strong bisimulation is a congruence on algebras $\mathcal{A}(P)$, and therefore the quotient algebra $\mathcal{A}(P)/\leftrightarroweq$ is well-defined.

THEOREM 5.12 *Let $P = (\Sigma, R)$ be a De Simone system over A, let C be a Σ-context with variables in $\{x_i \mid i \in \mathsf{N}\}$, and let ξ be an evaluation in $\mathcal{A}(P)$. Then*

$$[\![C]\!]_{\mathcal{A}(P)}^{\xi} \leftrightarroweq \langle\!\langle C \rangle\!\rangle_P^{\xi \lceil var(C)}.$$

PROOF By induction on the structure of C. If C is of the form x_i, then

$$\langle\!\langle C \rangle\!\rangle_P^{\xi\lceil var(C)} = \langle\!\langle x_i \rangle\!\rangle_P(\xi(x_i)) = op(og(x_i, transducer(P)))(\xi(x_i)).$$

The operator graph $og(x_i, transducer(P))$ has a single state x_i, and all its transductions are of the form

$$x_i \xrightarrow[(i,a)]{a} x_i$$

for a in A. It follows that $op(og(x_i, transducer(P)))$ is the identity operation on $\widehat{\mathcal{G}}(A)$. This implies

$$op(og(x_i, transducer(P)))(\xi(x_i)) = \xi(x_i) = [\![C]\!]_{\mathcal{B}}^{\xi}.$$

If C is of the form $f(C_1, \ldots, C_n)$, then we derive (with $\mathcal{B}$ short for $\mathcal{A}(P)$):

$$[\![C]\!]_{\mathcal{B}}^{\xi} =$$

$$= \langle\!\langle f(x_1, \ldots, x_n) \rangle\!\rangle_P([\![C_1]\!]_{\mathcal{B}}^{\xi}, \ldots, [\![C_n]\!]_{\mathcal{B}}^{\xi}) \qquad \{\text{by Definitions 3.7 and 5.11}\}$$

$$\leftrightarrows \langle\!\langle f(x_1, \ldots, x_n) \rangle\!\rangle_P(\langle\!\langle C_1 \rangle\!\rangle_P^{\xi\lceil var(C_1)}, \ldots, \langle\!\langle C_n \rangle\!\rangle_P^{\xi\lceil var(C_n)}) \qquad \{\text{by ind.hyp. and Lemma 5.4}\}$$

$$\leftrightarrows \langle\!\langle C \rangle\!\rangle_P^{\xi\lceil var(C)} \qquad \{\text{by Lemma 5.10}\}$$

■

COROLLARY 5.13 (COMPOSITIONAL AND OPERATIONAL SEMANTICS AGREE) *Let P be a De Simone system and let t be a closed term over the signature of P. Then $[\![t]\!]_{\mathcal{A}(P)} \leftrightarrows \mathcal{O}_P(t)$.*

PROOF By combination of Lemma 5.9 and Theorem 5.12. ■

One possible interpretation of Corollary 5.13 and the counterexample that a similar result does not hold up to isomorphism, is that there is some arbitrariness in the definitions of $[\![]\!]_{\mathcal{A}(P)}$ and $\mathcal{O}_P()$. This arbitrariness disappears if one considers the resulting graphs up to strong bisimulation congruence.

Lemma 5.10 and Lemma 5.4 can be used to give a short proof of Theorem 3.11, which says that bisimulation equivalence is a congruence for De Simone calculi. Because, suppose $P = (\Sigma, A, R)$ is a De Simone calculus, C is a unary context over Σ, and t and t' are closed terms over Σ with $\mathcal{O}_P(t) \leftrightarrows \mathcal{O}_P(t')$. Then

$$\mathcal{O}_P(C[t]) \leftrightarrows$$

$$\leftrightarrows \langle\!\langle C[t] \rangle\!\rangle_P() \qquad \{\text{by Lemma 5.9}\}$$

$$\leftrightarrows \langle\!\langle C[x_1] \rangle\!\rangle_P(\langle\!\langle t \rangle\!\rangle_P()) \qquad \{\text{by Lemma 5.10}\}$$

$$\leftrightarrows \langle\!\langle C[x_1] \rangle\!\rangle_P(\mathcal{O}_P(t)) \qquad \{\text{by Lemmas 5.4 and 5.9}\}$$

$$\leftrightarrows \langle\!\langle C[x_1] \rangle\!\rangle_P(\mathcal{O}_P(t')) \qquad \{\text{by Lemma 5.4}\}$$

$$\leftrightarrows \cdots \leftrightarrows \mathcal{O}_P(C[t']).$$

Basically, what happens in the above derivation is that the question whether bisimulation is a congruence, is reduced via Lemma 5.10, from a problem in the *syntactic* world of SOS to a problem in the *semantic* world of action transducers. We claim that the same reduction can be used to give simple congruence proofs for a variety of behavioral equivalences which are coarser than bisimulation equivalence.

5.2 Realizing Operations in PC

DEFINITION 5.14 [Realizability] Let $\mathcal{A}$ be a Σ-algebra and let f be an n-ary operation on a subset D of $D_{\mathcal{A}}$. We say that f is *realizable* (or *definable*) in terms of the operations of $\mathcal{A}$ if there exists a term t over signature Σ with $var(t) = \{x_1, \ldots, x_n\}$ such that for all valuations $\xi : \mathcal{V} \to D$, $f(\xi(x_1), \ldots, \xi(x_n)) = [\![t]\!]^{\xi}_{\mathcal{A}}$.

The following theorem gives a sufficient condition for realizability in the setting of De Simone systems:

THEOREM 5.15 *Let P and Q be De Simone calculi over A, let f be an n-ary function symbol of P, and let C be a context over the signature of Q with variables $\{x_1, \ldots, x_n\}$, such that*

$$og(f(x_1, \ldots, x_n), transducer(P)) \leftrightarrows og(C, transducer(Q)).$$

Then $f_{\mathcal{A}(P)/\leftrightarrows}$ is realizable in $\mathcal{A}(Q)/\leftrightarrows$.

Once the notion of realizability has been defined, it is easy to see that also the other expressiveness result of De Simone [33, 34] depends in a crucial way on the use of unguarded recursion. First, we will state De Simone's theorem using the terminology of this paper.

As action alphabet De Simone considers an infinite commutative monoid M (The reader may just think of M as the set of natural numbers). In addition, a finite signature Σ is considered and a finite collection of rules of the form

$$\frac{\{x_i \xrightarrow{u_i} y_i \mid i \in I\}}{f(x_1, \ldots, x_n) \xrightarrow{u} t} Pr(u_{j_1}, \ldots, u_{j_l}, u)$$

where $I = \{j_1, \ldots, j_l\}$. These rules are De Simone rules in our sense, except that the $u_i, \ldots$ which occur above the arrows are variables ranging over actions and not actions. Moreover the rules have as an additional ingredient a recursively enumerable relation Pr on M. The reader may think of a rule in the above format as a way to define a *set* of rules in our sense of Definition 3.10, one for each instantiation of the action variables for which the predicate holds. In order to distinguish the above format from the De Simone format introduced earlier, we will refer to it as the *classic* De Simone format.

Phrased in the terminology of this paper, De Simone [33, 34] proved that any operation of the algebra induced by a specification in classic De Simone format (induced in the sense of Definition 5.11 with $\widehat{\mathcal{G}}(M)$ taken as domain) can be realized up to bisimulation equivalence in terms of the operations of the algebra induced by the calculi SCCS and MEIJE.

The question arises to what extent this result still holds if the guarded versions of SCCS and MEIJE are used. In guarded SCCS and MEIJE only finitely branching graphs can be specified. However, using the classic De Simone format it is easy to specify an infinitely

branching graph that is not bisimilar with any finitely branching graph: just take a constant ω with the single rule

$$\frac{\emptyset}{\omega \xrightarrow{u} \omega}\ true.$$

Thus some restrictions have to be imposed on the classic De Simone format if we want to maintain the expressiveness result in a guarded setting. An obvious restriction is to allow only for predicates $Pr(u_1, \ldots, u_l, u)$ with for each $a_1, \ldots, a_l \in M$ the set $\{a \in M \mid Pr(a_1, \ldots, a_l, a)\}$ finite and recursive (together with its cardinality). However, this does not work. It is trivial to check that the rules of the calculus $Triple$ in the proof of Theorem 4.9 fit the restricted format. Consider the result of applying the operation $triple_{\mathcal{A}(P)}$ on the graph $\mathcal{O}_{Triple}(1)$. Clearly, the resulting graph is isomorphic to the graph g defined in Theorem 4.9. However, as a corollary of Theorem 4.9, the graph g can not be specified up to trace equivalence in proces calculi like SCCS and MEIJE with guarded recursion. Thus the operation $triple_{\mathcal{A}(P)}$ is certainly not realizable up to bisimulation in terms of the operations of these calculi.

We can now state the following theorem, which asserts that the calculus PC is universally expressive for operations definable by finite De Simone systems.

THEOREM 5.16 *Let f be an operation on the domain of finitely branching process graphs over some finite alphabet A that is specified via a De Simone system with a finite number of rules. Then f is realizable in terms of the operations of (a finite instantiation of) PC.*

PROOF Similar to proof of the corresponding result in [34], using Theorem 5.15. ■

REFERENCES

[1] S. Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53:225–241, 1987.

[2] L. Aceto, B. Bloom, and F.W. Vaandrager. Turning SOS rules into equations. In *Proceedings 7^{th} Annual Symposium on Logic in Computer Science,* Santa Cruz, California, pages 113–124. IEEE Computer Society Press, 1992. Full version available as CWI Report CS-R9218, June 1992, Amsterdam. Invited to the LICS 92 Special Issue of *Information and Computation.*

[3] K.R. Apt and G.D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, October 1986.

[4] E. Badouel and P. Darondeau. Structural operational specifications and trace automata. In W.R. Cleaveland, editor, *Proceedings CONCUR 92,* Stony Brook, NY, USA, volume 630 of *Lecture Notes in Computer Science*, pages 302–316. Springer-Verlag, 1992.

[5] J.C.M. Baeten and J.A. Bergstra. Global renaming operators in concrete process algebra. *Information and Computation*, 78(3):205–245, 1988.

[6] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. On the consistency of Koomen's fair abstraction rule. *Theoretical Computer Science*, 51(1/2):129–176, 1987.

[7] J.C.M. Baeten and F.W. Vaandrager. An algebra for process creation. *Acta Informatica*, 29(4):303–334, 1992.

[8] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[9] G. Berry and G. Gonthier. The synchronous programming language **Esterel**: design, semantics, implementation. Report 842, INRIA, Centre Sophia-Antipolis, Valbonne Cedex, 1988. To appear in *Science of Computer Programming.*

[10] B. Bloom. *Ready Simulation, Bisimulation, and the Semantics of CCS-like Languages.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, August 1989.

[11] B. Bloom. Strong process equivalence in the presence of hidden moves. Preliminary report, October 1990.

[12] B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be traced: Preliminary report. In *Conference Record of the* 15^{th} *ACM Symposium on Principles of Programming Languages,* San Diego, California, pages 229–239, 1988. Full version available as Technical Report 90-1150, Department of Computer Science, Cornell University, Ithaca, New York, August 1990. Accepted to appear in *Journal of the ACM.*

[13] R.N. Bol and J.F. Groote. The meaning of negative premises in transition system specifications (extended abstract). In J. Leach Albert, B. Monien, and M. Rodríguez, editors, *Proceedings* 18^{th} *ICALP,* Madrid, volume 510 of *Lecture Notes in Computer Science*, pages 481–494. Springer-Verlag, 1991. Full version appeared as Report CS-R9054, CWI, Amsterdam, 1990.

[14] G. Boudol. Notes on algebraic calculi of processes. In K. Apt, editor, *Logics and Models of Concurrent Systems*, pages 261–303. Springer-Verlag, 1985. NATO ASI Series F13.

[15] P. Darondeau. Concurrency and computability. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes, Proceedings LITP Spring School on Theoretical Computer Science,* La Roche Posay, France, volume 469 of *Lecture Notes in Computer Science*, pages 223–238. Springer-Verlag, 1990.

[16] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[17] R.J. van Glabbeek and U. Goltz. Refinement of actions in causality based models. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness,* Mook, The Netherlands 1989, volume 430 of *Lecture Notes in Computer Science*, pages 267–300. Springer-Verlag, 1990.

[18] J.F. Groote. Transition system specifications with negative premises. Report CS-R8950, CWI, Amsterdam, 1989. An extended abstract appeared in J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90*, Amsterdam, LNCS 458, pages 332–341. Springer-Verlag, 1990.

[19] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, October 1992.

[20] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, Cambridge, Massachusetts, 1988.

[21] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1985.

[22] K.G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. In M. Paterson, editor, *Proceedings* 17^{th} *ICALP,* Warwick, volume 443 of *Lecture Notes in Computer Science*, pages 526–539. Springer-Verlag, July 1990. An extended version appeared as: Report R89-13, The University of Aalborg, Dept. of Mathematics and Computer Science, Aalborg, Denmark, May 1989.

[23] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.

[24] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[25] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.

[26] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, 5^{th} *GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[27] J. Parrow. The expressive power of parallelism. *Future Generation Computer Systems*, 6:271–285, 1990.

[28] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[29] G.D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Proceedings IFIP TC2 Working Conference on Formal Description of Programming Concepts – II,* Garmisch, pages 199–225, Amsterdam, 1983. North-Holland.

[30] A. Ponse. Computable processes and bisimulation equivalence. Report CS-R9207, CWI, Amsterdam, January 1992.

[31] H. Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill Book Co., 1967.

[32] J.J.M.M. Rutten. Deriving denotational models for bisimulation from structured operational semantics. In M. Broy and C.B. Jones, editors, *Proceedings IFIP Working Conference on Programming Concepts and Methods,* Sea of Gallilea, Israel, pages 155–177. North-Holland, 1990.

[33] R. de Simone. *Calculabilité et Expressivité dans l'Algebra de Processus Parallèles* MEIJE. Thèse de 3^e cycle, Univ. Paris 7, 1984.

[34] R. de Simone. Higher-level synchronising devices in MEIJE–SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[35] F.W. Vaandrager. On the relationship between process algebra and input/output automata (extended abstract). In *Proceedings* 6^{th} *Annual Symposium on Logic in Computer Science,* Amsterdam, pages 387–398. IEEE Computer Society Press, 1991.