On the Expressiveness of ACP

(extended abstract)*

Rob van Glabbeek†
Computer Science Department, Stanford University
Stanford, CA 94305, USA
`rvg@cs.stanford.edu`

De Simone showed that a wide class of languages, including CCS, SCCS, CSP and ACP, are expressible up to strong bisimulation equivalence in Meije. He also showed that every recursively enumerable process graph is representable by a Meije expression. Meije in turn is expressible in aprACP (ACP with action prefixing instead of sequential composition).

Vaandrager established that both results crucially depend on the use of unguarded recursion, and its noncomputable consequences. *Effective* versions of CCS, SCCS, Meije and ACP, not using unguarded recursion, are incapable of expressing all effective De Simone languages. And no effective language can denote all computable process graphs.

In this paper I recreate De Simone's results in aprACP without using unguarded recursion. The price to be payed for this is the use of a partial recursive communication function and—for the second result—a single constant denoting a simple infinitely branching process. Due to the noncomputable communication function, the version of aprACP employed is still not effective.

However, I also define a wide class of De Simone languages that are expressible in an effective version of aprACP. This class includes the effective versions of CCS, SCCS, ACP, Meije and most other languages proposed in the literature, but not CSP. An even wider class, including CSP, turns out to be expressible in an effective version of aprACP to which an effective relational renaming operator has been added.

1 Introduction

In the early 1980's several languages for the description of communicating processes were introduced, most notably CCS [8], SCCS [9], CSP [5] and ACP [3]. Using these languages for purposes of specification and verification showed the necessity, or at least the convenience, of adding many additional constructs tailored to specific applications. In foundational research however, for instance when proving a property by structural induction, it is more convenient to have a fixed and well-defined set of operators. This consideration gives rise to the task of finding a basic language in which all or most useful operators can be expressed. Although the languages CCS and SCCS were designed with this goal in mind, the language Meije, proposed by Austry & Boudol [1], was the first result of a systematic analysis of expressiveness issues. In [13], Robert

---

[*] To appear in A. Ponse, C. Verhoef & S.F.M. van Vlijmen, editors: Proceedings ACP94, Workshop on Algebra of Communicating Processes, Utrecht, The Netherlands, May 1994, *Workshops in Computing*, Springer-Verlag, 1994.

[†] This work was supported by ONR under grant number N00014-92-J-1974.

de Simone observed that all constructs of the languages CCS, SCCS, CSP and ACP, and most constructs that are used in specific applications, can be defined in a particular way—namely by structural operational rules that fit in what is now known as the *De Simone format*—and showed that any operator that can so be defined is expressible in Meije, up to strong bisimulation equivalence. I will refer to a language all of whose constructs can so be defined as a *De Simone language.*

The interleaving semantics of CCS-like languages is most conveniently described in terms of *process graphs* or *labelled transition systems.* Depending on how constructive one wants the theory to be, several classes of graphs or transition systems can be considered, as indicated in Figure 1. For any uncountable cardinal κ one can define the *κ-bounded* process graphs to be the ones with less than κ (reachable) nodes (states) and edges (transitions), or—clearly equivalent—the ones in which every node has less than κ outgoing edges. For $\kappa = \aleph_0$, however, these definitions would not be equivalent. The *regular* process graphs (having a finite set of states and transitions) form a strictly smaller class than the *finitely branching* ones. Additionally one can consider the *recursively enumerable* graphs, whose nodes form a recursive set (such as the natural numbers), and whose edges form a recursively enumerable set. If, moreover, it is decidable whether a prospective edge is present or not, such a graph is *decidable.* Furthermore, a *computable* process graph is one for which there exists a terminating algorithm that, when given a node, produces as output the finite set of its outgoing edges. This notion was introduced in Baeten, Bergstra & Klop [2]. Note that it is a stronger requirement than "decidable and finitely branching". Finally, in Ponse [11] a graph is called *primitive recursive* if it is computable by means of a primitive recursive algorithm.

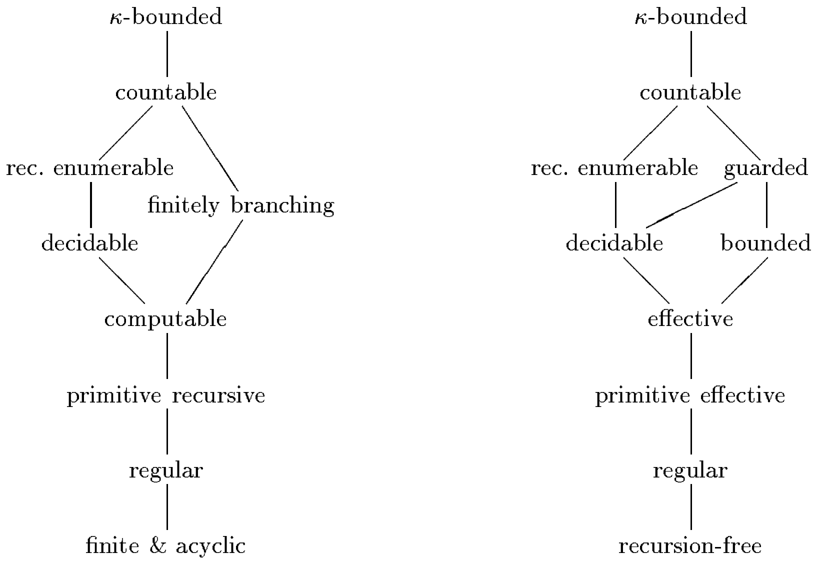

Figure 1: Classes of process graphs . . . and process expressions

In this paper I propose a similar classification for (expressions in) De Simone languages. For each class of process graphs in Figure 1 I introduce a class of (process) expressions that are guaranteed to denote graphs from that class only. The classes of process expressions are displayed in the second part of Figure 1. The notions of *recursion-free*, *regular*, and *κ-bounded* expressions are just mentioned for the sake of completeness, and are defined for CCS-like languages only. The *countable* process expressions are the ones in which all operators, not counting recursion, have finite arities. Except in the section on κ-bounded expressions, I presume this to be the default. The concepts *bounded* and *effective* are due to VAANDRAGER [14]. These two general but simple restrictions ensure that the effected expressions only denote, respectively, finitely branching and computable process graphs. Applied to CCS-like languages boundedness reduces to guardedness of recursive specifications. The notion of effectivity is also applied to languages as a whole. In that case it presupposes a decidable set of closed terms, and requires the interpretation which maps closed terms to descriptions of computable process graphs to be computable as well.

Whereas for languages like ACP and CCS one can define a κ-bounded version (containing only κ-bounded expressions), a recursively enumerable version, an effective version, etc., the language MEIJE was designed in such a way that all its constructs, except for recursion, are intrinsically effective. Nevertheless, as shown in DE SIMONE [12], many undecidable and infinitely branching processes can be specified by MEIJE-expressions. The expressive power needed to do so can only originate from the use of unguarded recursion. It follows that the restriction to guarded recursion must be a prerequisite in the definitions of decidable and bounded expressions. This is indicated in Figure 1. The language MEIJE with unguarded recursion falls in the class of recursively enumerable languages. In MEIJE only finite recursion is used.

In [13], DE SIMONE shows that MEIJE is universal in two different ways. It is universal among the De Simone languages, and it has a universal power to specify process graphs. Namely, any finitary recursively enumerable De Simone language is expressible in MEIJE, and any recursively enumerable process graph is representable by a MEIJE expression. Here *finitariness* is a syntactic restriction that is met by virtually all De Simone languages encountered in practice. Both expressiveness results hold up to bisimulation equivalence.

VAANDRAGER [14] investigates whether similar expressiveness results can be obtained for effective languages, and comes up with two negative results. First of all he points out that no effective language is capable of denoting all computable process graphs. This had been shown already by BAETEN, BERGSTRA & KLOP [2] up to strong bisimulation equivalence, but Vaandrager sharpens the result to hold for the coarser notion of trace equivalence as well. Secondly he defines a finitary effective De Simone language PC that cannot be translated in any effective version of MEIJE, CCS, SCCS or ACP with only finite guarded recursion. The culprit is a *relational renaming operator* that occurs in this language. By means of this operator a primitive recursive graph can be specified that is not specifiable in the mentioned languages.

The main contributions of the present paper are two effective versions of De Simone's first expressivity result. As unguarded recursion violates effectivity, its use must be eliminated from De Simone's construction. But MEIJE appears to lose most of its power when unguarded recursion is gone. Therefore I will use a variant of ACP [3], called aprACP$_F$. aprACP$_F$ is a parametrised language, of which an instantiation is obtained by selecting a set A of *actions* and a partial binary *communication function* on A. One particular choice of the communication function yields an extension of MEIJE. Thus all expressiveness results obtained for MEIJE hold for aprACP$_F$ as well. However, thanks to the possibility of other communication functions, aprACP$_F$ is more flexible, and potentially more expressive.

In Section 2 I propose a definition of what it means for one language to be expressible in another. This definition agrees with the notion of a *translation* from BOUDOL [4]. I also introduce (annotated) signatures to determine the syntax of a language and review the method of structural operational semantics for interpreting the closed expressions in a language as (equivalence classes of) process graphs.

In Section 3 I introduce the language aprACP$_F$, and indicate how it relates to ACP, CCS and MEIJE. In short, it is a sublanguage of ACP—thus "apr" (for *action prefix*)—to which renaming operations have been added—thus the subscript F (for *functional renaming*). CCS as well as MEIJE are obtained as sublanguages of aprACP$_F$ under particular instantiations of the parameters. I show that several operators, such as the *left-merge*, the *communication merge* and the *alternative composition*, that play a rôle in ACP, are expressible in terms of the other operators, and hence need not to be introduced as primitives. I also show that one can benefit from all choices of communication functions at the same time. There is namely one canonical instantiation of aprACP$_F$ in which any other instantiation with the same set of actions can be expressed.

In Section 4 I define the classification of aprACP$_F$-expressions of Figure 1. When possible, I make these classes of expressions large enough to denote every graph in the corresponding class. Only in case of the decidable aprACP$_F$-expressions I do not succeed in this. In particular, each computable process graph can be represented by an effective expression. This had been shown earlier by PONSE [11] for the language μCRL. However, it is not possible to combine enough such expressions in one effective language, as follows from the first negative expressiveness result mentioned above. When one tries to do this, the set of closed terms of the language becomes undecidable. For the same reason I couldn't find a canonical candidate for an effective version of aprACP$_F$, in which all other (finitary) effective versions of aprACP$_F$ can be expressed. This leaves not much hope for expressing all finitary effective De Simone languages in aprACP$_F$. Therefore I settle for the *primitive effective* De Simone languages, denoting only primitive recursive graphs. By means of a trivial construction, every such graph can be denoted in a single primitive effective version of aprACP$_F$. As the construction uses infinite—but primitive recursive—guarded recursion, this is still not a primitive effective version of De Simone's second expressiveness result. The same can be said for a more disciplined construction by PONSE [11].

In Section 5 I extend the classification of Figure 1, or actually the part between "countable" and "primitive effective", to arbitrary De Simone languages. Subsequently I state the expressiveness results that I obtained, but in this extended abstract the proofs are not included.

Vaandrager's counterexample-language PC needs only primitive recursion, so not all finitary primitive effective De Simone languages translate in aprACP$_F$ with finite guarded recursion. Therefore I isolate a class that do. These are the *functional* finitary primitive effective De Simone languages. They include the primitive effective versions of CCS, SCCS, ACP and MEIJE. The restriction to primitive recursion is probably not so bad, as I am not aware of any use of effective but not primitive effective De Simone languages. However, the requirement of functionality rules out CSP and Vaandrager's PC. This is solved by adding the relational renaming operator of PC—which to a great extent also occurs as the *(inverse) image operator* in CSP—to aprACP$_F$, thereby obtaining aprACP$_R$. All finitary primitive effective De Simone languages are expressible in the primitive recursive version of aprACP$_R$ using only finite guarded recursion. I do not know if these results carry over to MEIJE. They do not carry over to CCS, CSP or PC.

Besides these main contributions I establish similar results for the countable De Simone languages and the bounded ones. Again, I use only aprACP$_R$ with finite guarded recursion. Similarly, I recreate De Simone's first expressiveness result for aprACP$_R$, without using unguarded recursion. For this, an undecidable communication function is used.

All results announced above are also generalised to non-finitary De Simone languages. This necessitates the use of infinite, but still guarded, recursion.

VAANDRAGER [14] established that any finite effective De Simone language is expressible in a finite version of PC. Here *finite* means specified by means of a finite number of De Simone rules. Such languages only denote graphs with a finite set of actions. It should be noted that any finite De Simone language with guarded recursion is functional, finitary and primitive effective. Finite guarded De Simone languages can also be expressed in aprACP$_F$ and MEIJE.

As far as the power to specify process graphs goes, I recreate De Simone's result while using only finite guarded recursion. For this purpose I use a version of aprACP$_F$ with a partial recursive (but undecidable) communication function, to which a single constant U has been added denoting a simple infinitely branching process. Without adding U no infinitely branching processes can be specified in aprACP$_F$ with guarded recursion. This result did not extend to other classes of process graphs/expressions.

2 Syntax, Semantics and Expressibility

In this section I propose a definition of what it means for one language to be expressible in another. In order for such a notion of expressibility to be meaningful, a language is understood to combine syntax and semantics.

First I will introduce the syntax of a language through the notion of a *annotated signature*. The annotated signature determines what are the valid expressions of the language. I will define a signature as a set of function declarations, thus omitting the possibility of predicates. These do not occur in the languages I want to study. The annotation specifies to what extent recursion is incorporated in the language.

The semantics of a language is given through an interpretation of the (closed) terms in a domain of values. Such an interpretation should be *compositional* and satisfy a few other requirements. My notion of a compositional semantics generalises notions of *denotational semantics*, namely by not insisting that the meaning of recursive expressions is obtained by order-theoretic methods. Thus my concept of expressiveness applies to languages with a denotational semantics as well.

Subsequently I treat the notion of Structural Operational Semantics in PLOTKIN's style [10]. The languages considered in this paper will all be equipped with a structural operational semantics. Such an operational semantics associates in a standard way a *process graph* to every closed process expression.

On process graphs, I divide out *bisimulation equivalence*, which is among the finest congruence relations available. All my expressibility results will be established up to bisimulation equivalence. They will then also hold for most other—less fine—equivalences.

2.1 Syntax

In this paper V is an infinite set of *variables*, ranged over by X, X_i, x, y, x' etc.

Definition 1 (*Signatures*). A *function declaration* is a pair (f,n) of a *function symbol* $f \notin V$ and an *arity* $n \in \mathbb{N}$. A function declaration $(c,0)$ is also called a *constant declaration*. A *signature* is a set of function declarations. The set $\mathbb{T}^r(\Sigma)$ of *terms with recursion* over a signature Σ is defined inductively by:

- $V \subseteq \mathbb{T}^r(\Sigma)$,
- if $(f,n) \in \Sigma$ and $t_1, \ldots, t_n \in \mathbb{T}^r(\Sigma)$ then $f(t_1, \ldots, t_n) \in \mathbb{T}^r(\Sigma)$,
- If $V_S \subseteq V$, $S : V_S \rightarrow \mathbb{T}^r(\Sigma)$ and $X \in V_S$, then $\langle X|S\rangle \in \mathbb{T}^r(\Sigma)$.

A term $c()$ is often abbreviated as c. A function S as appears in the last clause is called a *recursive specification*. A recursive specification S is often displayed as $\{X = S_X \mid X \in V_S\}$. An occurrence of a variable y in a term t is *free* if it does not occur in a subterm of the form $\langle X|S\rangle$ with $y \in V_S$. A term is *closed* if it contains no free occurrences of variables. Let $T^r(\Sigma)$ be the set of closed terms over Σ. The sets $\mathbb{T}(\Sigma)$ and $T(\Sigma)$ of open and closed terms over Σ without recursion are defined likewise, but without the last clause.

The syntax of a language can be given as a signature together with an annotation which places some restrictions on the use of recursion. These can be:
κ-bounded: the sets S should have cardinality less than κ,
countable: the sets S should be countable,
enumerable: the functions S should be partial recursive,
computable: the sets V_S should be decidable and the functions S recursive,
primitive: as above, but using only primitive recursion,
finite: the sets S should be finite,
guarded: the sets of equations S should satisfy a syntactic criterion that ensures that they have unique solutions under a given interpretation,
or no recursion.

Definition 2 (*Substitutions*). A Σ-*substitution* σ is a partial function from V to $\mathbb{T}^r(\Sigma)$. If σ is a substitution and t a term, then $t[\sigma]$ denotes the term obtained from t by replacing, for x in the domain of σ, every free occurrence of x in t by $\sigma(x)$, while renaming bound variables if necessary to prevent name-clashes. In that case $t[\sigma]$ is called a *substitution instance* of t. A substitution instance $t[\sigma]$ where σ is given by $\sigma(x_i) = s_i$ for $i \in I$ is denoted as $t[s_i/x_i]_{i\in I}$. These notions extend to syntactic objects containing terms, with the understanding that such an object is a substitution instance of another one only if the same substitution has been applied to each of its constituent terms.

2.2 Expressibility

A *language* can be given by an annotated signature, specifying its syntax, and an *interpretation*, assigning to every (closed) term t its meaning $[\![t]\!]$. The meaning of a closed term can simply be a *value* chosen from a set of values $\mathbb{D}$, which is called a *domain*. Usually interpretations are required to satisfy some sanity requirements. One of them is that the meaning of a term $\langle X|S\rangle$ is the X-component of a solution of S. To be precise:

$$[\![\langle X|S\rangle]\!] = [\![S_X[\langle Y|S\rangle/Y]_{Y\in V_S}]\!].$$

Other requirements are *compositionality* and *invariance under α-recursion*. Compositionality demands that the meaning of a term is completely determined by the meaning of its components. This means that for functions $(f,n)\in\Sigma$ and for recursive specifications S and S' with $X\in V_S = V_{S'}$ we have

$$\begin{aligned} [\![t_i]\!] = [\![t_i']\!]\ (i=1,...,n) \quad &\Rightarrow \quad [\![f(t_1,...,t_n)]\!] = [\![f(t_1',...,t_n')]\!] \\ \text{and } [\![S_Y]\!] = [\![S_Y']\!]\ (Y\in V_S) \quad &\Rightarrow \quad [\![\langle X|S\rangle]\!] = [\![\langle X|S'\rangle]\!]. \end{aligned}$$

Invariance demands that the meaning of a term is independent of the names of its bound variables, i.e. for any injective substitution $\alpha : V_S \rightarrow V$

$$[\![\langle\alpha(X)|S[\alpha]\rangle]\!] = [\![\langle X|S\rangle]\!].$$

In order for language L_1 to be expressible in language L_2, I require that for every closed L_1-term t there exists a closed L_2-term $\tilde{t}$ denoting the same value. Usually this can only be the case if the domain $\mathbb{D}_1$ in which L_1-expressions are interpreted is included in the domain $\mathbb{D}_2$ of L_2.

The requirement on closed terms is not sufficient. It says that every value denotable by language L_1 can also be denoted by language L_2. In addition I want that every *operation* of L_1 can be mimicked in L_2. This has to do with the meaning of open terms. For every open term in $\mathbb{T}^r(\Sigma_1)$ I want to find a term in $\mathbb{T}^r(\Sigma_2)$ with the same meaning.

A common approach to open terms is to reduce them to the collection of their closed substitution instances. In this view there is no need to extend the interpretation $[\![\cdot]\!]$ explicitly to open terms. The preconditions $[\![S_Y]\!] = [\![S_Y']\!]$ of the second compositionality requirement are simply read as "$[\![S_Y[\sigma]]\!] = [\![S_Y'[\sigma]]\!]$ for each closed substitution $\sigma : V \rightarrow T^r(\Sigma)$". This approach is often taken when generalising an equivalence relation on closed terms to an equivalence on open terms (over the same signature). Two open terms are then declared equivalent if for every closed substitution the corresponding substitution instances are equivalent.

It is slightly more difficult to employ this approach in defining expressibility. The problem is that open terms over different signatures are compared, so that it is impossible to employ the same substitution at both sides. This is solved as follows:

Definition 3 (*Expressibility*) Let L_i $(i=1,2)$ be two languages, given as an annotated signature Σ_i and an interpretation $[\![\cdot]\!]_i : T^r(\Sigma_i) \rightarrow \mathbb{D}_i$. L_1 is said to be *expressible* in L_2 if there is a translation $\tilde{\ } : \mathbb{T}^r(\Sigma_1) \rightarrow \mathbb{T}^r(\Sigma_2)$ such that for all $t\in\mathbb{T}^r(\Sigma_1)$ with free variables $x_1,...,x_n$ and for all $t_1,...,t_n \in T^r(\Sigma)$

$$[\![t[t_i/x_i]_{i=1}^n]\!]_1 = [\![\tilde{t}[\tilde{t}_i/x_i]_{i=1}^n]\!]_2.$$

This will be my definition of expressibility for now. In the full version of this paper, however, I plan to be a bit more ambitious. An open term with n free variables is interpreted as an n-ary operator on the domain $\mathbb{D}$, and for L_1 to be expressible in L_2 I require that for every L_1-term t there is an L_2-term $\tilde{t}$, such that t and $\tilde{t}$ denote the same operator, at least when applied to values from the domain of L_1. If L_1 can be expressed into L_2 in that sense, it can certainly be done in the sense of Definition 3.

2.3 Quotient domains

Let $\sim$ be an equivalence relation on a domain $\mathbb{D}$. An interpretation $[\![\cdot]\!]$ in $\mathbb{D}$ is *compositional up to* $\sim$ if it satisfies the requirements for compositionality, but with "=" replaced by "$\sim$". The first requirement for instance reads

$$[\![t_i]\!] \sim [\![t'_i]\!] \ (i = 1, ..., n) \quad \Rightarrow \quad [\![f(t_1, ..., t_n)]\!] \sim [\![f(t'_1, ..., t'_n)]\!].$$

In the same way the other sanity requirements, as well as the notion of expressibility, can be defined *up to* $\sim$. In the case of expressibility, however, it is necessary that $\sim$ is defined on $\mathbb{D}_1 \cup \mathbb{D}_2$.

Given a domain $\mathbb{D}$ for interpreting languages and an equivalence relation $\sim$, the *quotient domain* $\mathbb{D}/_\sim$ consists of the $\sim$-equivalence classes of elements of $\mathbb{D}$. An interpretation $[\![\cdot]\!] : T^R(\Sigma) \rightarrow \mathbb{D}$ of the closed terms of a language in $\mathbb{D}$, is turned into the *quotient interpretation* $[\![\cdot]\!]_\sim : T^R(\Sigma) \rightarrow \mathbb{D}/_\sim$ of these terms by letting $[\![t]\!]_\sim$ be the equivalence class containing $[\![t]\!]$. This quotient interpretation satisfies the sanity requirements of Section 2.2 iff the original interpretation satisfies them up to $\sim$. Likewise, one language is expressible in another under a quotient interpretation obtained by dividing out the same equivalence $\sim$ on their domains of interpretation, iff, under the original interpretation, this language is expressible in the other up to $\sim$.

2.4 Process Graphs

When the expressions in a language are meant to represent processes, they are called *process expressions*, and the language a *process description language*. Suitable domains for interpreting process description languages are the class of *process graphs* and its quotients. In such *graph domains* a process is represented by either a process graph, or an equivalence class of process graphs. Process graphs are also known as *state-transition diagrams* or *automata*. They are *labelled transition systems* equipped with an initial state.

Definition 4 (*Process graphs*) A *process graph*, labelled over a set A of actions, is a triple $G = (S, T, I)$ with

- S a set of *nodes* or *states*,
- $T \subseteq S \times A \times S$ a set of *edges* or *transitions*,
- and $I \in S$ the *root* or *initial* state.

Let $\mathbb{G}(A)$ be the domain of process graphs labelled over A.

Virtually all so-called *interleaving models* for the representation of processes are isomorphic to graph models. The *failure sets* for instance that represent expressions in the process description language CSP [5] can easily be exchanged for equivalence classes of graphs, under a suitable equivalence. In [3] the language ACP is equipped with a process graph semantics, and the semantics of CCS, SCCS and MEIJE given in [8, 9, 1, 13] are operational ones, which, as I will show below, induce a process graph semantics.

Usually the parts of a graph that cannot be reached from the initial state by following a finite path of transitions are considered meaningless for the description of processes. This means that one is only interested in process graphs as a model of system behaviour up to some equivalence, and this equivalence identifies at least graphs with the same reachable parts. Likewise, the particular identity of the states in a process graph is normally not of any importance. Two graphs that only differ in the naming of their states are called *isomorphic* and also isomorphic graphs are semantically identified.

Definition 5 (*Reachability*). Let $G = (S, T, I)$ be a process graph. A *path* is an alternating sequence $s_0, a_1, s_1, a_2, s_2, \ldots, s_{n-1}, a_n, s_n$ of states and actions, such that $s_{i-1} \xrightarrow{a_i} s_i$ for $i = 1, \ldots, n$. Here $s_{i-1} \xrightarrow{a_i} s_i$ is an abbreviation for $(s_{i-1}, a_i, s_i) \in T$. Such a path is said *to go from* s_0 *to* s_n. A state s' is *reachable* from a state s if there is a path from s to s'. The *reachable part* of G is the graph (S^R, T^R, I) with $S^R \subseteq S$ the set of states reachable from the initial state I, and $T^R = T \cap (S^R \times A \times S^R)$.

(*Isomorphism*). Two process graphs $G = (S, T, I)$ and $H = (S', T', I')$ are *isomorphic* if there exists a bijection $f : S \rightarrow S'$—called an *isomorphism*—with $f(I) = I'$ and $(s, a, t) \in T \Leftrightarrow (f(s), a, f(t)) \in T'$.

Write $G \cong H$ if the reachable parts of G and H are isomorphic.

Thus $\cong$ may be considered the finest equivalence (the one with the fewest identifications, and the smallest equivalence classes) on $\mathbb{G}$ that makes $\mathbb{G}/_{\cong}$ in a reasonable model of concurrency. However, some languages (interpretations) encountered in this paper fail to be compositional up to $\cong$. Also several expressiveness results will not hold up to $\cong$. Therefore a coarser equivalence (identifying more, and having larger equivalence classes) should be divided out on $\mathbb{G}$. In the literature many equivalences have been proposed. The finest of those is (strong) bisimulation equivalence, due to MILNER [8, 9].

Definition 6 (*Bisimulation equivalence*). Two process graphs G and H are *bisimulation equivalent*—notation $G \leftrightarroweq H$—if there exists a binary relation R—called a *bisimulation*—between their states, such that

- the initial states of G and H are related,
- if sRt and $s \xrightarrow{a} s'$ then H has a state t' with $t \xrightarrow{a} t'$ and $s'Rt'$,
- if sRt and $t \xrightarrow{a} t'$ then G has a state s' with $s \xrightarrow{a} s'$ and $s'Rt'$.

All languages mentioned in this paper satisfy the sanity requirements of Section 2.2 up to $\leftrightarroweq$. The expressiveness results of this paper will also be established up to bisimulation equivalence. This means that they surely hold for the (coarser) equivalences used elsewhere in the literature, such as *weak bisimulation equivalence*—the standard equivalence of CCS [8]—and (weak) failures equivalence—the standard semantics of CSP [5].

2.5 Operational Semantics

In this section I present PLOTKIN's method of *Structural Operational Semantics* [10] for interpreting expressions as process graphs or labelled transition systems.

Definition 7 (*Transition system specifications;* GROOTE & VAANDRAGER [6]). Let Σ be an annotated signature and A a set (of *actions*). A *(positive)* (Σ, A)-*literal* is an expression $t \xrightarrow{a} t'$ with $t, t' \in \mathbb{T}^r(\Sigma)$ and $a \in A$. An *action rule* over (Σ, A) is an expression of the form $\frac{H}{\alpha}$ with H a set of (Σ, A)-literals (the *premises* of the the rule) and α a (Σ, A)-literal (the *conclusion*). A rule $\frac{H}{\alpha}$ with $H = \emptyset$ is also written α. A *transition system specification (TSS)* is a triple (Σ, A, R) with Σ a signature and R a set of action rules over (Σ, A).

The following definition tells when a transition is provable from a TSS. It generalises the standard definition (see e.g. [6]) by (also) allowing the derivation of rules. The derivation of a transition $t \xrightarrow{a} t'$ corresponds to the derivation of the rule $\frac{H}{t \xrightarrow{a} t'}$ with $H = \emptyset$. The case $H \neq \emptyset$ corresponds to the derivation of $t \xrightarrow{a} t'$ under the assumptions H.

Definition 8 (*Proof*). Let $P = (\Sigma, R)$ be a TSS. A *proof* of an action rule $\frac{H}{\alpha}$ from P is a well-founded, upwardly branching tree of which the nodes are labelled by Σ-literals, such that:

- the root is labelled by α, and
- if β is the label of a node q and K is the set of labels of the nodes directly above q, then
  - either $K = \emptyset$ and $\beta \in H$,
  - or $\frac{K}{\beta}$ is a substitution instance of a rule from R.

If a proof of $\frac{H}{\alpha}$ from P exists, then $\frac{H}{\alpha}$ is *provable* from P, notation $P \vdash \frac{H}{\alpha}$.

Transition system specifications often contain infinitely many rules, yet are presented finitely by giving *rule schemata*, each of which codifies a large set of rules. This practice is formalized in part by the notion of an abstract TSS.

Definition 9 (*Abstract TSSs*). An *abstract* Σ-*literal* is an expression $t \longrightarrow t'$ with $t, t' \in \mathbb{T}(\Sigma)$. An *abstract action rule* over (Σ, A) is an expression of the form $\frac{H,\ Pr}{\alpha}$ with H a set of abstract (Σ, A)-literals, $Pr \subseteq A^H \times A$, and α an abstract Σ-literal. An *abstract TSS* is a triple (Σ, A, R) with Σ a signature and R a set of abstract action rules over (Σ, A). An abstract TSS (Σ, A, R) *determines* the (concrete) TSS (Σ, A, R') with

$$R' = \left\{ \frac{\{t_i \xrightarrow{a_i} t'_i \mid i \in I\}}{t \xrightarrow{b} t'} \;\middle|\; \frac{\{t_i \longrightarrow t'_i \mid i \in I\},\ Pr}{t \longrightarrow t'} \in R \wedge Pr(\vec{a}, b) \right\}.$$

Finally I will show how the operational semantics of a language, given as a TSS, induces a process graph semantics.

Definition 10 (*Interpreting the closed expressions in a TSS as process graphs*). Let $P = (\Sigma, A, R)$ be a TSS and $t \in T(\Sigma)$. Then $[\![t]\!]$ is defined to be the reachable part of the process graph $(T^r(\Sigma), T, t)$ with T the set of transitions provable from P.

3 Prefix ACP with Relational Renaming

The language that I will use for my expressiveness results is a variant of ACP [3] that could be called *prefix ACP with relational renaming.* Like ACP this language has two parameters: an alphabet A of *actions* and a partial *communication function* $| : A^2 \to A$, which is commutative and associative, i.e.

- $a|b = b|a$ (commutativity)
- $(a|b)|c = a|(b|c)$ (associativity)

for all $a, b, c \in A$ (and each side of these equations is defined just when the other side is). I will denote this language as $\text{aprACP}_R(A, |)$.

Its signature contains a constant 0 denoting inaction, two binary operators $+$ and $\|$ denoting *alternative* and *parallel composition* respectively, a unary operator a for any action $a \in A$, a unary *encapsulation* operator ∂_H for any $H \subseteq A$ and a *relational renaming* operator ρ_R for any binary relation $R \subseteq A \times A$.

$p\|q$ represents the independent execution of the processes p and q, partly synchronized by the communication function $|$. If $a|b$ is defined, an occurrence of a in p can synchronise with an occurrence of b in q into a communication action $a|b$ between p and q. If $a|b$ is not defined, no such communication is possible. The action a of p can, instead of synchronising with an action of q, (also) appear independent of q, and likewise can b occur independently of p.

The process ap first performs the action $a \in A$ and then behaves like p. $\partial_H(p)$ behaves like p, but without the possibility of performing actions from H. The operator ρ_R is a slight generalisation of the relabelling and (inverse) image operators of CCS and CSP. 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 one action b that is related to a via R.

In $\text{aprACP}_R(A, |)$-expressions brackets are omitted under the convention that a binds strongest and $+$ weakest. Besides aprACP with relation renaming I also consider aprACP with functional renaming, denoted $\text{aprACP}_F(A, |)$. This is the same language, but with a renaming operator ρ_f only for every *function* $f : A \to A$ instead of for every relation.

The action rules for $\text{aprACP}_R(A, |)$ are given in Table 1, thereby completing the formalisation of this language as a TSS. These rules determine an interpretation of the $\text{aprACP}_R(A, |)$-expressions in $\mathbb{G}(A)$. This interpretation agrees, up to bisimulation equivalence, with the more denotational interpretation of $\text{ACP}(A, |)$ in $\mathbb{G}(A)$ given in [Baeten,] Bergstra & Klop [3, 2].

It is common to regard the entries in tables like 1 as schemata, each of which denotes a rule for any proper instantiation of the metavariables a, b, c by real actions from A. Thus in case A is infinite, there are infinitely many rules for every operator. In an attempt at "finitisation" I will in this paper regard each of the first six entries as single rules of an abstract TSS. The rule $\dfrac{x \xrightarrow{a} x'}{x\|y \xrightarrow{a} x'\|y}$ for instance, should be read as $\dfrac{x \longrightarrow x',\ Id}{x\|y \longrightarrow x'\|y}$ where $Id \subseteq A \times A$ is the identity relation on A. In the rules were Pr is not the identity, it is explicitely given. The last three entries in Table 1 remain schemata even when interpreted as abstract action rules. There is namely one rule for every encapsulation operator, one for every relational renaming, and one for every pair $\langle X|S\rangle$ with S a recursive specification and $X \in V_S$. But at least there are now finitely many rules for every operator, even if A is infinite. This will turn out to be a useful property.

$ax \xrightarrow{a} x$	$\dfrac{x \xrightarrow{a} x'}{x+y \xrightarrow{a} x'}$	$\dfrac{y \xrightarrow{a} y'}{x+y \xrightarrow{a} y'}$
$\dfrac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y}$	$\dfrac{x \xrightarrow{a} x',\ y \xrightarrow{b} y',\ a \vert b = c}{x \parallel y \xrightarrow{c} x' \parallel y'}$	$\dfrac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'}$
$\dfrac{x \xrightarrow{a} x',\ a \notin H}{\partial_H(x) \xrightarrow{a} \partial_H(x')}$	$\dfrac{S_x[\langle Y \vert S\rangle / Y]_{Y \in V_S} \xrightarrow{a} z}{\langle X \vert S\rangle \xrightarrow{a} z}$	$\dfrac{x \xrightarrow{a} x',\ R(a,b)}{\rho_R(x) \xrightarrow{b} \rho_R(x')}$

Table 1: aprACP_R

3.1 CCS

MILNER's Calulus of Communicating Systems (CCS) [8] can be regarded as (a sublanguage of) an instantiation of aprACP with functional renaming. CCS is parametrised with a set $\mathcal{A}$ of *names*. The set $\bar{\mathcal{A}}$ of *co-names* is given by $\bar{\mathcal{A}} = \{\bar{a} \mid a \in \mathcal{A}\}$, and $\mathcal{L} = \mathcal{A} \cup \bar{\mathcal{A}}$ is the set of *labels*. The function $\bar{\cdot}$ is extended to $\mathcal{L}$ by declaring $\bar{\bar{a}} = a$. Finally $Act = L \cup \{\tau\}$ is the set of *actions*. CCS can now be presented as aprACP$(Act, |)$, where $|$ is the partial function on Act given by $a|\bar{a} = \tau$.

In CCS there is a renaming operator only for every function $f : Act \rightarrow Act$ that satisfies $f(\bar{a}) = \overline{f(a)}$ and $f(\tau) = \tau$. This operator (applied on a process p) is written $p[f]$ and called *relabelling*. Also there is an encapsulation operator ∂_H only when $H = A \cup \bar{A}$ with $A \subseteq \mathcal{A}$. This operator is called *restriction* and is written $p \backslash A$. Parallel composition is written $|$ instead of $\|$, but there is no further difference between CCS and apr$\mathrm{ACP}_F(Act, |)$.

3.2 ACP

There are many methodological differences between the ACP approach to *process algebra* and the CCS approach. I will not address these here. As a language, ACP can be regarded as a modification of CCS in four directions.

- First of all, ACP makes a distinction between deadlock and successful termination. As a consequence, action prefixing can be replaced by action constants and a general sequential composition.
- ACP adds two auxiliary operators, the *left merge* and the *communication merge*, denoted $\lfloor\!\lfloor$ and $|$, to enable a finite equational axiomatization of the parallel composition.
- Whereas CCS combines communication and abstraction from internal actions in one operator, in ACP these activities are separated. In CCS the result of any communication is the unobservable action τ. In ACP it is an observable action, from which (in the extended language ACP_τ) one can abstract by applying an *abstraction* operator, renaming designated actions into τ.

- CCS adheres to a specific communication format, admitting only handshaking communication, whereas ACP allows a variety of communication paradigmas, including ternary communication, through the choice of the communication function $|$.

In this paper only the last feature of ACP is of importance. I don't distinguish observable and unobservable actions and therefore work with ACP rather than ACP$_\tau$. As I also don't deal with the distinction between deadlock and successful termination, I restrict attention to the sublanguage aprACP of ACP that doesn't make this distinction and consequently supports prefixing only. Whereas in the original papers on ACP the set A of action constants was required to be finite, I allow it to be infinite. I add the subscript R (or F) to indicate the addition of (functional) renaming operators, which were not included in the syntax of ACP. Subsequently I drop the auxiliary operators $\mathbin{|\!\!\lfloor}$ and $|$ from the language, since these can be expressed in the other operators of aprACP$_F$, as I will show in Section 3.5. The resulting language extends CCS in only one essential way, namely through the general communication format. This generality greatly enhances the expressiveness of the language.

3.3 Meije

Like CCS, also BOUDOL's language MEIJE [1, 4, 13] can be regarded as (a sublanguage of) an instantiation of aprACP with functional renaming. MEIJE is parametrised with a set $\mathcal{A}$ of *atomic actions* and a set $\mathcal{S}$ of *signals*. The set *Act* of *actions* is a commutative monoid, namely the free commutative product of the free commutative moniod generated by $\mathcal{A}$ and the free commutative group generated by $\mathcal{S}$. This means that the elements of *Act* are a kind of multisets over $\mathcal{A}$ and $\mathcal{S}$ with the stipulation that elements of $\mathcal{S}$ may also have negative multiplicities. These can also be seen as ordinary multisets over $\mathcal{A}$, $\mathcal{S}$ and $\mathcal{S}^{-1} = \{s^{-1} \mid s \in \mathcal{S}\}$ in which s and s^{-1} cancel. A typical element of *Act* is denoted as $a^5b^2s^3t^{-1}$. The product operation . on *Act* is such that $a^5b^2s^3t^{-1}.a^2c^4s^{-3}t^3u^{-3} = a^7b^2c^4t^2u^{-3}$. MEIJE can now be presented as a sublanguage of aprACP$_F(Act, .)$.

In MEIJE there are two kind of renaming operators. There is a renaming operator ρ_Φ, written $\langle\Phi\rangle(p)$, for any *morphism* $\Phi : Act \rightarrow Act$. Here a morphism is a function satisfying $\Phi(a.b) = \Phi(a).\Phi(b)$ for $a, b \in Act$. In addition there is a renaming operator $s * p$, called *ticking*, for any signal $s \in S$. This operator renames any action a into $s.a$. The only type of encapsulation operators permitted in MEIJE are the restriction operators $p\backslash s$ for any signal $s \in S$. $p\backslash s$ is $\partial_H(p)$ for H the set of all actions containing s, i.e. all "multisets" in which s has a positive or negative multiplicity. MEIJE also has an operator *triggering* which is expressible in the others, and it lacks the operator $+$, because, as explained in Section 3.6, that operator is expressible in the others as well, but there is no further difference between MEIJE and aprACP$_F(Act, .)$.

3.4 A Decidable Signature for aprACP

The language aprACP$_R(A, |)$ defined so far has an uncountable signature if A is infinite. There are namely uncountably many encapsulation and renaming operators. Computationally it makes sense to restrict attention to a fragment of aprACP$_R$ with a decidable signature. This can be achieved by requiring the set of actions A to be decidable, and by restricting the permitted encapsulation and renaming operators ∂_H and ρ_R to the ones where $A - H$ and R are recursively enumerable sets. Such sets can be represented by the code of a turing machine that enumerates them, and it is decidable whether an arbitrary piece of text is the code of a turing machine enumerating a recursive enumarable set. This makes the signature decidable. As a consequence the set of recursion-free terms will be decidable too.

I could have chosen H to be enumerable instead of its complement $A - H$. However, the choice above has the advantage that the encapsulation operators can (in an obvious way) be regarded as special relational renamings, so that one has one kind of operator less to be concerned about. A more compelling argument will be presented in Section 4.4.

In order to ensure that the set of recursive terms is decidable as well, I have to require recursive specifications, seen as sets of equations, to be recursively enumerable at least. This makes the set of open terms decidable. However, it remains undecidable whether a term is closed. The set of closed terms is not even enumerable.

Therefore one may wish to insist that in terms of the form $\langle X|S\rangle$ the set of recursion variables V_S is decidable as well. This makes S computable. However, it is undecidable whether a piece of text is the code of a turing machine deciding membership of a set. Thus with computable recursion even the set of open terms becomes undecidable again.

Hence an even more restrictive requirement on the desired kind of recursion is in order. Here I require S to be primitive decidable. This means that there is a primitive recursive function deciding membership of V_S, and in case of a variable $X \in V_S$ returning the term S_X. It is decidable whether a piece of text is the source of a primitive recursive function, thus with this restriction the signature as well as the sets of open and closed terms are decidable.

If moreover the communication function is required to be partial recursive, the resulting variant of aprACP$_R$ will be denoted aprACP$_R^{\text{r.e.}}$. The other languages I mentioned can be adapted in the same way. In aprACP$_F^{\text{r.e.}}$ I have to allow partial recursive renaming functions. In the original version of aprACP$_F$, these where expressible in terms of total renamings and encapsulation.

3.5 Expressing the Left- and Communication Merge

The language ACP has two operators, $\|\!\_$ and $|$, that I didn't include in the syntax of aprACP$_F$. The reason is that these operators can be expressed in the other operators of the language, and thus need not be introduced as primitives. Here I show how. Table 2 shows the action rules for the two operators. The

$$\boxed{\frac{x \xrightarrow{a} x'}{x \,\|\!\_\, y \xrightarrow{a} x' \| y} \qquad \frac{x \xrightarrow{a} x',\ y \xrightarrow{b} y',\ a|b = c}{x \mid y \xrightarrow{c} x' \| y'}}$$

Table 2: The left- and communication merge of ACP

left merge, $\mathbin{\|\!\!\_}$, behaves exactly like the *merge* or parallel composition, $\|$, except that the first action is required to come from its leftmost argument. The communication merge, $|$, behaves exactly like $\|$, except that its first action is required to be a communication between its two arguments. The operators' most crucial use is in the axiom CM1

$$x\|y = x \mathbin{\|\!\!\_} y + y \mathbin{\|\!\!\_} x + x \mid y$$

that plays an essential rôle in axiomatising ACP with bisimulation semantics.

Note that the symbol $|$ is used for the communication function as well as the communication merge. This overloading is intentional, as the communication merge can be thought of as an extension of the communication function, which is defined on actions only. The vertical bar in the middle of an expression $\langle X|S\rangle$ is pronounced *where*—and sometimes even written that way—and has nothing to do with the communication function and merge. The vertical bar in a set expression like $\{n \in \mathbb{N} \mid n > 5\}$ is also pronounced *where* and constitutes a fourth use of this symbol. Finally $|$ is used to denote parallel composition in CCS. It is generally easy to determine from the context which $|$ is meant.

In order to express $\mathbin{\|\!\!\_}$ and $|$ in aprACP$_F(A, |)$ I assume that the set of actions A is divided into a set A_0 of actions that may be encountered in applications, and the remainder $H_0 = A - A_0$, which is used as a working space for implementing useful operators, such as $\mathbin{\|\!\!\_}$ and $|$. On A_0 the communication function is dictated by the applications, but on H_0 I can choose it in any way that suits me. The cardinality of A_0 should be infinite, and equal to the cardinality of A.

For today's implementation I assume that H_0 contains actions **`skip`**, **`first`**, **`next`**, $a_{\texttt{first}}$ and $a_{\texttt{next}}$ (for $a \in A_0$). The communication function given on A_0 is extended to these actions as indicated below (applying the convention that if $a|b$ is not defined it is undefined).

$a \mid \texttt{first} = a_{\texttt{first}}$	$(a \in A_0)$
$a \mid \texttt{next} = a_{\texttt{next}}$	$(a \in A_0)$
$a_{\texttt{first}} \mid b_{\texttt{first}} = a \mid b$	$(a, b \in A_0)$
$a \mid \texttt{skip} = a$	$(a \in A_0)$

Let $H_1 = A_0 \cup \{\texttt{first}, \texttt{next}\}$. I will use a renaming operator f_1 that satisfies $f_1(a_{\texttt{next}}) = a$, $f_1(a_{\texttt{first}}) = a_{\texttt{first}}$ and $f_1(a) = a$ for $a \in A_0$.

Let me first introduce the notation a^∞ to denote a process that perpetually performs the action a. This process is obtained as $a^\infty = \langle X \mid X = aX\rangle$. Now suppose p is a process that can do actions from A_0 only. Then

$$\partial_{H_1}(p\|\texttt{first}(\texttt{next}^\infty))$$

is a process that behaves exactly like p, except that every initial action has a tag (subscript) **`first`**, and every non-initial action has a tag **`next`**. Thus $\rho_{f_1}(\partial_{H_1}[p\|\texttt{first}(\texttt{next}^\infty)])$ is a process that behaves exactly like p, except that the initial actions are tagged **`first`**. It follows that for any two processes p and q with actions from A_0 only

$$p \mid q \mathrel{\underline{\leftrightarrow}} \partial_{H_0}\Big(\rho_{f_1}(\partial_{H_1}[p\|\texttt{first}(\texttt{next}^\infty)])\Big\|\rho_{f_1}(\partial_{H_1}[q\|\texttt{first}(\texttt{next}^\infty)])\Big)\,.$$

In order to extend this result to processes with actions outside A_0 I use a bijective renaming $f_0 : A \rightarrow A_0$ and its inverse f_0^{-1}. The communication merge is expressed in $\mathrm{aprACP}_F(A, |)$, up to bisimulation equivalence, by

$$x \mid y \mathrel{\underline{\leftrightarrow}} \rho_{f_0^{-1}} \left(\partial_{H_0} \left(\rho_{f_1}(\partial_{H_1}[\rho_{f_0}(x) \| \mathtt{first}(\mathtt{next}^\infty)]) \middle\| \rho_{f_1}(\partial_{H_1}[\rho_{f_0}(y) \| \cdots])\right)\right).$$

Finally the left merge is expressed in terms of the communication merge through

$$x \mathbin{\underline{\|}} y \mathrel{\underline{\leftrightarrow}} \rho_{f_0^{-1}}(\mathtt{skip}(\rho_{f_0}(y)) \mid \rho_{f_0}(x)).$$

3.6 Expressing Choice

As remarked in Section 3.3, the language MEIJE lacks the choice operator + of CCS and ACP. The reason this operator was omitted from the syntax of MEIJE was that it can be expressed in terms of the other operators. This is true in the setting of aprACP_F as well, so if one likes, this operator can be skipped from the signature.

For the implementation of choice, A is again divided in A_0 and H_0 and in H_0 we put the same actions as in the previous section, together with the action `choose`. The communication function also works as before, except that there is no communication possible between actions of the form $a_{\mathtt{first}}$ and $b_{\mathtt{first}}$. (So maybe one wants to use a different action `first`.) Instead one has the communication $\mathtt{choose} \mid a_{\mathtt{first}} = a$ for $a \in A_0$. Recall that for p a process that can do actions from A_0 only, $\rho_{f_1}(\partial_{H_1}[p \| \mathtt{first}(\mathtt{next}^\infty)])$ is the same process in which the initial actions are tagged with a subscript `first`. This, by the way, is an implementation of the operator *triggering* of MEIJE. It follows that

$$p + q \mathrel{\underline{\leftrightarrow}} \partial_{H_0} \left(\rho_{f_1}(\partial_{H_1}[p \| \mathtt{first}(\mathtt{next}^\infty)]) \middle\| \mathtt{choose} \middle\| \rho_{f_1}(\partial_{H_1}[q \| \mathtt{first}(\mathtt{next}^\infty)])\right)$$

since in the expression on the right `choose` can communicate with an initial action from only one of p or q, so that the other one is blocked forever. Using the renamings ρ_{f_0} and its inverse, just as in the previous section, + is expressed in the rest of aprACP_F.

3.7 Expressing the Communication Function

It may be felt as a drawback that the language (apr)ACP, unlike CCS, is parametrised by the choice of a communication function (besides the choice of a set of actions). This, one could argue, makes it into a collection of languages rather then a single one. Personally I do not share this concern. If in different applications different communication functions are used, they can, when desired, all be regarded as different fragments of the same communication function, each considered on only a small subset of the set of actions A. At any given time there is no need to know all actions and the entire communication function to be used in all further applications.

Alternatively one may argue that there are many parallel compositions possible in ACP, namely one for every choice of a communication function. Here I will present one instantiation of $\mathrm{aprACP}_F(A, |)$, such that for every other choice of a communication function the resulting parallel composition is expressible in this language.

Let, as in the previous section, $A_0 \subseteq A$ be the actions that are used in applications and $H_0 = A - A_0$ the working space. I fix a bijection $f_0 : A \rightarrow A_0$ and abbreviate ρ_{f_0} by ρ_0. For this implementation, H_0 should contain the actions (a, b) for every $a, b \in A_0$, as well as an action δ denoting deadlock. The communication function $|$ is defined by $a|b = (a, b)$ for $a, b \in A_0$ (thus undefined outside A_0). Now for any other communication function $\gamma : A^2 \rightarrow A$ let $\bar{\gamma} : A \rightarrow A$ be a renaming satisfying $\bar{\gamma}((\rho_0(a), \rho_0(b))) = c$ for those $a, b, c \in A$ with $\gamma(a, b) = c$, and $\bar{\gamma}(a) = f_0^{-1}(a)$ for $a \in A_0$. Furthermore, let H be $\{(a, b) \in A_0 \times A_0 \mid \gamma(a, b) \text{ undefined}\}$. Then the associated parallel composition $\|_\gamma$ is expressible in aprACP$_F(A, |)$ by

$$x\|_\gamma y \mathrel{\underline{\leftrightarrow}} \rho_{\bar{\gamma}}(\partial_H(\rho_0(x)\|\rho_0(y))).$$

3.8 Expressing Renaming and Encapsulation

The syntax of aprACP$_F^{\text{r.e.}}$ allows for a multitude of renaming operators. Here I show that one needs only two, namely the operator ρ_0, introduced earlier, which bijectively maps every action to one in the subset A_0 of A, and the *universal renaming operator* ρ_F. Every other functional renaming is then expressible.

To this end I introduce an action $\bar{f} \in H_0 = A - A_0$ for every partial recursive renaming function $f : A \rightarrow A$. I also introduce an action $(\bar{f}, a) \in H_0$ for every such f and every $a \in A_0$. The communication function is enriched by $\bar{f} \mid a = (\bar{f}, a)$ for $a \in A_0$ and the universal renaming F should satisfy $F((\bar{f}, \rho_0(a))) = f(a)$. Let $H_1 = A_0 \cup \{\bar{f} \mid f : A \rightarrow A\}$. Then $\partial_{H_1}(\bar{f}^\infty\|\rho_0(x))$ is a process that behaves exactly like x, except that every action a is renamed in $(\bar{f}, \rho_0(a))$. Hence ρ_f is expressible through $\rho_f(x) \mathrel{\underline{\leftrightarrow}} \rho_F(\partial_{H_1}(\bar{f}^\infty\|\rho_0(x)))$.

Note that every co-enumerable encapsulation operator can be regarded as a partial recursive renaming, so also all encapsulation operators can be expressed in aprACP with ρ_0 and ρ_F. The encapsulation ∂_{H_1} used in the construction can be incorporated in ρ_F as well.

In exactly the same way every enumerable relational renaming operator is expressible in aprACP with only ρ_0 and a *universal relational renaming.*

In the preceding section I showed how all operators $\|_\gamma$ with γ a communication function could be expressed in a particular instantiation of aprACP$_F$, using a multitude of renamings. Here I showed how all renaming operators of aprACP$_F^{\text{r.e.}}$ can be expressed in only two of them, using a particular communication function, and similarly for the encapsulations. It is an easy exercise to combine these results, and express all partial recursive renaming operators, all co-enumerable encapsulations, and all parallel compositions $\|_\gamma$ with γ a partial recursive communication function in a particular instantiation of aprACP$_F^{\text{r.e.}}$, with only one communication function and two renamings.

One may wonder whether *all* renaming operators can be expressed in aprACP, i.e. if it is possible to get rid of the last two. In general this is not possible. However, if one only cares about the behaviour of all the derived operators on the relevant subset A_0 of A, it is possible to omit the use of ρ_0 from all the constructions, encode the universal renaming in the communication function, and find for any derived operator (such as $\mathbin{|\!\lfloor}$ or a renaming) an aprACP-expression with the same behaviour on A_0.

3.9 A Finite Signature for aprACP

Here I show how the syntax of $\mathrm{aprACP}_R^{\mathrm{r.e.}}$ can be reduced from a decidable one to a finite one. In the previous section the set of renaming and encapsulation operators was cut down to two elements, so we are left with an infinity of actions to get rid of. As in $\mathrm{aprACP}_R^{\mathrm{r.e.}}$ the set of actions is decidable, they can be numbered $a_0, a_1, a_2, \ldots$, such that the function $\mathtt{succ} : A \to A$ given by $\mathtt{succ}(a_i) = a_{i+1}$ is partial recursive (even computable). Hence every action is expressible in terms of a_0 and the renaming operator $\rho_{\mathtt{succ}}$.

3.10 Expressing Relational Renaming

In order to express relational renamings in aprACP_F one needs to add just one constant (or a third renaming) to the signature. One has to assume the existence of actions $[a,b]$ in H_0 for $a, b \in A_0$. The desired constant is $\mathtt{all} = \Sigma_{a,b \in A_0}[a,b]0$. Here $\Sigma_{i \in I}$ is an infinite version of choice, to be formally introduced in Section 4.1. $\Sigma_{i \in I}$ is not a standard ingredient in the syntax of aprACP—if it were there would be no reason to add `all` as a constant. `all` can be expressed as $\rho_R(c0)$ where c is an action chosen from A and R is the relation $\{(c, [a,b]) \mid a, b \in A_0\}$.

Now $\mathtt{allever} = \langle X \mid X = \mathtt{all} \mathbin{\|\!\!\_} X \rangle$ is a process that perpetually performs an action of the form $[a,b]$, and at each step has the choice between all such actions. For any relation $R \subseteq A \times A$ the process $\partial_{A \times A - R}(\mathtt{allever})$ has at each step the choice between executing one the actions $[a,b]$ with $R(a,b)$. Let $\mathtt{copy} : A_0 \to A$ be a renaming that sends each action $a \in A_0$ to the (new) action $a_{\mathtt{copy}}$. Define the communication function on the new actions by $a_{\mathtt{copy}} \mid [a,b] = b$. Then for p a process that does actions from A_0 only

$$\rho_R(p) \;\underline{\leftrightarrow}\; \partial_{H_0}(\rho_{\mathtt{copy}}(p) \| \partial_{A \times A - R}(\mathtt{allever}))$$

Thus, by means of ρ_0 and its inverse to deal with action from outside A_0, all relational renaming operators are expressible in aprACP_F with `all`.

`all` can also be expressed in aprACP_F using so-called *unguarded recursion* (Section 4.3) as $\mathtt{all} = \langle X \mid X = [a_0, a_0]0 + \mathtt{succ2}(X) \rangle$ where `succ2` is a renaming function enumerating the elements of $A \times A$. Thus, as long as unguarded recursion is permitted, aprACP_F is equally expressive as aprACP_R. When unguarded recursion is banned, however, aprACP_R turns out to be more expressive.

4 Specifying Process Graphs

In this section I will isolate, for each of the classes of process graphs mentioned in the introduction, a corresponding class of process expressions that denote only graphs from that class. When possible, I make these classes of expressions so large that every graph of the appropriate kind can be denoted by an expression in the corresponding class.

Definition 11 (*Kinds of graphs*). A process graph $G = (S, T, I) \in \mathbb{G}(A)$ is

- *κ-bounded* (for an uncountable cardinal κ) if for every state $s \in S$ there are less than κ outgoing transitions $s \xrightarrow{a} s'$,

- *countable* if for every state $s \in S$ there are at most countably many outgoing transitions $s \xrightarrow{a} s'$,
- *finitely branching* if for every state $s \in S$ there are only finitely many outgoing transitions $s \xrightarrow{a} s'$,
- *recursively enumerable* if there exists an algorithm enumerating all transitions $s \xrightarrow{a} s'$,
- *decidable* if there exists an algorithm that, when given a triple $(s, a, s') \in S \times A \times S$, determines whether this is a transition from T,
- *computable* if there exists an algorithm that, when given a state $s \in S$, returns the complete finite list of outgoing transitions $s \xrightarrow{a} s'$ and indicates when the list is complete,
- *primitive recursive* if there is such an algorithm that is primitive recursive,
- and *regular* if it has only finitely many states and transitions.

The class of all expressions defined so far will be the class of countable process expressions. The κ-bounded process expressions are obtained by enlarging the signature, whereas the other classes of Figure 1 are obtained by means of restriction.

4.1 The κ-bounded Process Expressions

In order to define the κ-bounded process expressions I have to generalise the syntax of aprACP$_R$, even beyond the boundaries imposed by Definition 1. Whenever I is an index set and p_i are process expressions for $i \in I$, $\Sigma_{i \in I} p_i$ is now a process expression too. It represents a choice between the processes p_i $(i \in I)$. Since choice is associative and commutative for virtually every semantics proposed in the literature, I may be chosen to range over sets rather than sequences. The corresponding action rules (one abstract rule for every index set I and index $j \in I$) are

$$\frac{x_j \xrightarrow{a} y}{\Sigma_{i \in I} x_i \xrightarrow{a} y}.$$

For this purpose the set of variables should at least have cardinality κ. The expression $p_1 + p_2$ can now be regarded as an abbreviation for $\Sigma_{i=1,2} p_i$ and 0 as the summation over an empty index set. In the same fashion it is possible to introduce infinitary parallel compositions.

Now the *κ-bounded process expressions* can be defined as the ones in which all index sets, as well as the sets V_S in recursive specifications S, have cardinality less than κ. Also, for every relational renaming ρ_R and $a \in A$, the set $\{b \mid R(a, b)\}$ should have less than κ elements. It is straightforward to prove that process graphs associated to κ-bounded process expressions are κ-bounded. The converse is true as well:

Proposition 1 Every κ-bounded process graph can up to isomorphism be denoted by a κ-bounded process expression.

Proof: Let $G = (S, T, I)$ be a κ-bounded process graph. Take a variable X_s for every state $s \in S$ and let $\widetilde{G}$ be the recursive specification $\{X_s = \Sigma_{(s \xrightarrow{a} t) \in T} a X_t \mid s \in S\}$. Now $\langle X_I | \widetilde{G} \rangle$ is a closed process expression with $[\![\langle X_I | \widetilde{G} \rangle]\!] \cong G$. □

4.2 The Countable Process Expressions

In the special case of the countable process expressions ($\kappa = \aleph_1$) one allows countable alternative and parallel compositions and countable recursion. However, countable compositions can be expressed in terms of binary compositions and countable unguarded recursion. Namely

$$\Sigma_{i\in\mathbb{N}} p_i = \langle X_0 \mid \{X_i = p_i + X_{i+1} \mid i \in \mathbb{N}\}\rangle.$$

Thus the countable process expressions can be redefined to be the ones with countable recursion but only binary alternative and parallel composition.

Continuing from that perspective it can be observed that the addition of arbitrary infinite recursion does not add to the expressive power of the language, since in any expression $\langle X|S\rangle$ only countably many variables are reachable from X. Thus the countable process expressions can again be redefined to be exactly the ones introduced in Section 3. It follows that

Proposition 2 Countable process expressions yield countable process graphs and every countable process graph is denoted by a countable process expression.

4.3 The Bounded Process Expressions

The notion of guardedness was proposed in MILNER [8] to syntactically isolate a class of recursive specifications that have unique solutions. The definition below stems from BAETEN, BERGSTRA & KLOP [2].

Definition 12 (*Guardedness*). A free occurrence of a variable in a process expression t is *unguarded* if it does not occur in a subterm of the form at'. Let S be a recursive specification. The relation $\xrightarrow{u} \subseteq V_S \times V_S$ is given by $X \xrightarrow{u} Y$ iff Y occurs unguarded in S_X. S is *guarded* if the relation $\xrightarrow{u}$ is well-founded.

Besides ensuring unique solutions, the same requirement also helps to keep the denoted process graphs finitely branching. Let a process expression be *guarded* if in all its subexpressions $\langle X|S\rangle$ the recursive specification S is guarded.

Proposition 3 Guarded expressions in the languages (apr)ACP$_F$, CCS, SCCS and MEIJE denote finitely branching process graphs. Moreover, every finitely branching process graph is denoted by a guarded process expression.

Proof: The first statement follows with a straightforward induction on the structure of terms, with in the case of recursion a nested induction on the length of chains $X_1 \xrightarrow{u} X_2 \xrightarrow{u} \cdots$.

The second statement follows immediately from the proof of Proposition 1, considering that unguarded recursion wasn't used there. In Proposition 2 unguarded recursion has to be used to replace infinite alternative compositions, but in order to denote finitely branching graphs this is unnecessary. □

Due to the relational renaming (or inverse image) operator this proposition does not hold for aprACP$_R$ and CSP. In case the set $\{b \mid R(a,b)\}$ is infinite, $\rho_R(a)$ denotes an infinitely branching process graph.

Let a relation $R \subseteq A \times A$ be *image-finite* if $\forall a \in A : \{b \mid R(a,b)\}$ is finite. Then the *bounded* process expressions can be defined as the ones with only guarded recursion and image-finite renaming operators. It follows that the bounded process expressions denote only finitely branching graphs.

4.4 The Recursive Enumerable Process Expressions

The *recursively enumerable* process expressions are the ones that appear in the language $\mathrm{aprACP}_R^{\mathrm{r.e.}}(A,|)$. This is the variant of aprACP_R with a decidable signature, introduced in Section 3.4. A has to be decidable and $|$ a partial recursive function. Moreover, only encapsulation operators ∂_H with H co-r.e. are allowed (i.e. the complement of H should be enumerable), and only renaming operators ρ_R with R enumerable. Finally recursive specifications S are required to be primitive recursive.

Proposition 4 Any process $\Sigma_{i\in I}p_i$ with $\{p_i \mid i \in I\}$ a recursive enumerable set of recursive enumerable process expressions is expressible in $\mathrm{aprACP}_R^{\mathrm{r.e.}}$.

Proof: A classic recursion theoretic theorem states that any non-empty r.e. set can be obtained as the image of a primitive recursive function. See e.g. Corollary 4.18 in MANIN [7]. It follows that $[\![\Sigma_{i\in I}p_i]\!] \leftrightarrows [\![\Sigma_{i\in\mathbb{N}}p(n)]\!]$ for certain primitive recursive function $p : \mathbb{N} \to$ (the closed $\mathrm{aprACP}_R^{\mathrm{r.e.}}$-expressions). Now use the construction from Section 4.2. □

Proposition 5 The r.e. process expressions denote exactly the r.e. graphs.

Proof: It is straightforward to enumerate (recursively) the valid proofs of transitions between $\mathrm{aprACP}_R^{\mathrm{r.e.}}$ expressions, and hence the transitions themselves. Note that for this to be true one needs the complement of H in ∂_H to be enumerable rather than H itself. This is the argument promised in Section 3.4.

The other direction follows immediately from the proof of Proposition 1, using Proposition 4. □

DE SIMONE [13] proved that in order to denote every r.e. process graph it is sufficient to use finite recursion only. However, whereas the Propositions 1–5 are rather trivial and only use inaction, action prefix, choice and recursion, De Simone's construction is more intricate and uses the entire syntax of MEIJE.

Theorem 1 Let A be an decidable set of actions. There exists an decidable set of signals S, such that for every r.e. process graph $G \in \mathbb{G}(A)$ there exists a closed r.e. expression t with finite recursion in $\mathrm{MEIJE}(A,S)$ for which $[\![t]\!] \leftrightarrows G$.

As MEIJE can be implemented in aprACP_F, this theorem implies that for every decidable set A there is a decidable set $A' \supseteq A$ and a r.e. communication function $|$ on A', such that every r.e. process graph can, up to bisimulation, be denoted by an expression in $\mathrm{aprACP}_F^{\mathrm{r.e.}}(A',|)$ with only finite recursion.

It follows immediately from Proposition 3 that the use of unguarded recursion is unavoidable in De Simone's result. Still, it is possible to limit such use to a minimum. Suppose A contains actions a_i and b_i for $i \in \mathbb{N}$. The process $\Sigma_{i\in\mathbb{N}}a_ib_i0$ can be obtained with unguarded recursion as $\langle X | X = a_0b_00 + \rho_f(X)\rangle$ in which f is the renaming with $f(a_i) = a_{i+1}$ and $f(b_i) = b_{i+1}$ for $i \in \mathbb{N}$. Adding this process as a constant U to the language aprACP_F—thereby obtaining aprACP_U—makes it possible to recreate De Simone's result without using unguarded recursion.

Theorem 2 Let A be a decidable set of actions. There exists an decidable set of actions A' and a partial recursive communication function $|$ on A', such that for every r.e. process graph $G \in \mathbb{G}(A)$ there exists a closed expression t with finite guarded recursion in the language $\mathrm{aprACP}_U^{\mathrm{r.e.}}(A',|)$ for which $[\![t]\!] \leftrightarrows G$.

Proof: The proof is a variation on the one of De Simone. Consider the r.e. process graphs over A with as nodes the natural numbers. Since I consider graphs up to bisimulation equivalence, I may assume that there is at most one edge between every two nodes. Such a graph G can be represented by an algorithm g that, when given a pair of nodes (i,j), runs for some time and returns a in case there is a transition (i,a,j). In case there is no transition from i to j it returns the value δ or runs forever.

Now let A' be the set of all such algorithms g, together with A, a special symbol δ denoting (dead)lock, and the actions $\texttt{to}_i$ and $\texttt{from}_i$ for $i \in \mathbb{N}$. Note that the set of algorithms (in a particular form, such as Turing machine code) of partial recursive functions is decidable, i.e. it is decidable whether an arbitrary piece of text constitutes such an algorithm, and hence an element of A'. Let the communication function $|$ be given by $g \mid \texttt{from}_i \mid \texttt{to}_j = g(i,j)$. [To be precise, in order to implement this in the ACP communication format I also need actions of the form $\texttt{fromto}_{ij}$, $g(i,\cdot)$ and $g(\cdot,j)$].

The next thing I need is the *left merge* operator $\mathbin{|\!\!\lfloor}$ from ACP, which, as we saw in Section 3.5, can be expressed in aprACP$_F$. By means of a renaming, the new constant U can be turned into $\texttt{flow} = \Sigma_{i\in\mathbb{N}}\texttt{to}_i\texttt{from}_i 0$. This process describes the flow of control through an arbitrary state. It says that when one enters state i, the next thing to do is leaving the same state. Using only guarded recursion I subsequently define the process $\texttt{control} = \texttt{flow} \mathbin{|\!\!\lfloor} \texttt{control}$. This process will be put in a context where in each step a $\texttt{from}$ action and a $\texttt{to}$ action synchronise. As a result, when the n^{th} synchronisation involves a $\texttt{to}_i$ action, denoting the arrival in state i, the next synchronisation involves a $\texttt{from}_i$, denoting departure from the same state. The choice of the $\texttt{to}$ action is not restricted (by the $\texttt{control}$ process). In order to initialise the process properly, and to allow a first synchronisation, I use the initialised control $C = \texttt{from}_0 \| \texttt{control}$, in which 0 is supposed to be the initial state.

Now a graph G is represented by the expression $\partial_H(g^\infty \| C)$. Here g^∞ is a shorthand for $\langle X \mid X = gX\rangle$, i.e. the process that repeatedly performs the action g, and $H = A' - A$. It is easy to see that $[\![\partial_H(g^\infty \| C)]\!] \cong G$. □

Corollary 1 Let A be a countably infinite decidable set of actions. There exists a partial recursive communication function $|$ on A, such that for every r.e. process graph $G \in \mathbb{G}(A)$ there exists a closed expression t with finite guarded recursion in the language aprACP$_U^{\text{r.e.}}(A, |)$ for which $[\![t]\!] \underline{\leftrightarrow} G$.

Proof: Partition A into two infinite decidable subsets A_0 and H_0. It suffices to prove the statement for $G \in \mathbb{G}(A_0)$, since an arbitrary r.e. process graph can be obtained as $\rho_f(G)$ for such a G with $f: A_0 \to A$ a bijective renaming.

By Theorem 2 there is an extension A' of A_0 and a partial recursive communication function $|'$ on A', such that for every r.e. process graph $G \in \mathbb{G}(A_0)$ there exists a closed expression t with finite guarded recursion in the language aprACP$_U^{\text{r.e.}}(A', |')$ for which $[\![t]\!] \underline{\leftrightarrow} G$. Let $h: A' \to A$ be a recursive bijection with $h(a) = a$ for $a \in A_0$. Define the partial recursive communication function $| : A^2 \to A$ by $h(a) \mid h(b) = h(c)$. Let $\tilde{t}$ be the closed aprACP$_U^{\text{r.e.}}(A, |)$-expression obtained from t by replacing all action names a by $h(a)$, including the action names in the subscripts of encapsulation and renaming operators. Then $[\![t]\!] \underline{\leftrightarrow} G$ immediately implies $[\![\tilde{t}]\!] \underline{\leftrightarrow} G$. □

4.5 The Decidable and the Effective Expressions

De Simone [12] shows that with unguarded recursion it is easy to specify undecidable processes. Therefore the first requirement of a *decidable* process expression is that only guarded recursion is permitted. In this setting there are already three different variants of aprACP to consider: the language aprACP$_F$ with functional renaming, the language aprACP$_R$ with relational renaming, and the language aprACP$_U$ with the constant U, introduced in the previous section. The subscript U reminds of *universal* and is inspired by Theorem 2. Note that the guarded version of aprACP$^{\text{r.e.}}_U$ is at least as expressive as the guarded version of aprACP$^{\text{r.e.}}_R$. Namely, as A is decidable, the constant `all` of Section 3.10 can be obtained from U by means of encapsulation and renaming, and aprACP$^{\text{r.e.}}_R$ turned out to be guardedly expressible in terms of aprACP$^{\text{r.e.}}_F$ with `all`. As long as unguarded recursion was allowed, the three languages were equally expressive, but this is here no longer the case.

A second requirement for decidable process expressions has to do with the computable nature of the operators of the language. For the relational renaming operators ρ_R it is not sufficient to require the relations R to be decidable, as any recursively enumerable relation can be obtained as the composition of two decidable ones. In particular, the process $\Sigma_{a\in A}a0$ is surely decidable, and can be obtained as the image of a single action under a suitable decidable relational renaming. However, for any nonempty recursive enumerable set of actions $B \subseteq A$, the (generally undecidable) process $\Sigma_{b\in B}b0$ can be obtained as $\rho_f(\Sigma_{a\in A}a0)$ with f a primitive recursive function (recalling that a primitive recursive function is a special total recursive function, and any total recursive function, seen as a relation, is decidable). This is Corollary 4.18 in Manin [7].

A similar problem arises for the communication function. There are two ways in which to strengthen the decidability requirement so as to avoid these problems. The renaming operators as well as the communication function should be either *effective* or *coeffective*. The requirement of effectivity comes, in the more general setting of De Simone languages, from Vaandrager [14].

Definition 13 (*Decidable terms*) An aprACP$_R(A, |)$-expression is *effective* if

- A is a decidable set,
- $|$ is given as a total recursive function $| : A^2 \rightarrow A \dot{\cup} \{\delta\}$ —$a|b = \delta$ means that a and b do not communicate,
- it contains only computable guarded recursion,
- it contains only encapsulation operators ∂_H for which H is decidable
- and only renamings ρ_R for R such that $\forall a \in A : (\{b \mid R(a,b)\}$ is finite), and the total function which yields for any $a \in A$ this finite set is recursive.

A aprACP$_U(A, |)$-expression is *coeffective* if

- A is a decidable set,
- $|$ satisfies $\forall c \in A : (\{(a,b) \mid a|b = c\}$ is finite), and the total function which yields for any $c \in A$ this finite set is recursive,
- it contains only computable guarded recursion,
- it contains only encapsulation operators ∂_H for which H is decidable
- and only renamings ρ_R for R such that $\forall b \in A : (\{a \mid R(a,b)\}$ is finite), and the total function which yields for any $b \in A$ this finite set is recursive.

A aprACP$_U(A, |)$ expression is *decidable* if it is either effective and without the constant U or coeffective.

Note that, due to the use of countable recursion, the (co)effective expressions are not a subclass of the enumerable ones. Using only primitive recursive recursion would be a more serious restriction than in the the previous section, as Proposition 4 crucially depends on the use of unguarded recursion.

Mixing effective and coeffective ingredients in one process expression leads in general to undecidable processes. As indicated above, adding U to the effective processes is already catastrophic.

Proposition 6 Effective process expressions denote only computable graphs. Decidable expressions denote only decidable graphs.

Proof: Two straightforward structural inductions. □

I have no idea how to represent *all* decidable graphs by decidable process expressions. The proof strategy adopted for Theorem 2 makes use of a highly non-coeffective communication function as well as the non-effective constant U. Similarly I don't know if it is possible to represent all computable graphs by effective process expressions with finite recursion. However, the same recipe as used in the previous sections yields

Proposition 7 Every computable process graph is denoted by an effective process expression, using only choice, (in)action and recursion.

Proof: If $G \Subset \mathbb{G}(A)$ is computable it follows immediately that the recursive specification $\bar{G}$ constructed in the proof of Proposition 1 is computable. □

In Ponse [11], every computable graph is denoted by an effective expression in the language μCRL. The trivial proof above could be seen as a considerable simplification of his construction. However, Ponse uses finite recursion schemata, parametrised with recursive data parameters, whereas I use plain infinite recursion.

It is interesting to compare this positive result with the following negative one.

Definition 14 (*Effectivity*) An interpretation of a decidable language in a graph domain is *effective* if it induces a total recursive function from the closed terms to (descriptions of) computable process graphs.

The concept of an effective interpretation is due to Vaandrager [14]. Let a language be *decidable* if its set of closed terms is decidable (as in apr$\text{ACP}_{R}^{\text{r.e.}}$). The following theorem stems from Baeten, Bergstra & Klop [2]. Is has been sharpened in Vaandrager [14], who established it for the even coarser notion of trace equivalence.

Theorem 3 Let A be a set of at least two actions. No decidable language with an effective interpretation in $\mathbb{G}(A)$ is able to denote all computable process graphs, up to bisimulation equivalence.

If follows that the set of effective process expressions is not decidable. This is due to the presence of computable recursion, which was observed to make the language undecidable already in Section 3.4. Similarly, the set of expressions in Ponse's language is undecidable. It also follows that the decidability requirement in the above theorem is essential.

4.6 The Primitive Effective Process Expressions

In Section 3.4 I proposed a variant aprACP$_R^{\text{r.e.}}$ of aprACP$_R$ with a decidable signature. In Section 4.4 this language turned out to denote only enumerable process graphs. Furthermore in Section 3.7 there turned out to be a *canonical* instantiation of this language, such that any other instantiation (with a different communication function) is expressible in it. Here I search for a similar variant of aprACP in which only decidable graphs can be denoted.

The first idea could be to take the language of all effective process expressions in aprACP$_U^{\text{r.e.}}$ (or the coeffective ones or both). However, such a language has an undecidable signature. To be precise, as it is undecidable whether a piece of text is the code of a Turing machine representing a total recursive function, the set of renaming (and encapsulation) operators is undecidable.

Thus the collection of permitted encapsulation and renaming operators has to be cut down until their codes form a decidable set. It is tempting to think that Section 3.8 offers a solution, as it allows all these operators to be expressed in only two of them. However, the construction enabling this requires an action to be introduced in A for any renaming operator. This only shifts the undecidable signature problem from the renaming operators to the set of actions.

Another idea is allow any decidable selection of (co)effective renaming operators to constitute a valid instantiation of the desired language. This is basically what is done in VAANDRAGER [14] for the recursive specifications. There only one computable recursive specification is allowed. However, this violates the desired property of canonicity, as one cannot express all (co)effective renamings (or computable specifications) in one decidable selection.

Hence a (complexity) class of such operators has to be found for which it is decidable whether (a description of) an operator is in this class. As in Section 3.4 I take the class of *primitive recursive* operators. It should be admitted that many other (complexity) classes would serve the purpose equally well.

In order to maintain the canonicity result of Section 3.7, I also have to require primitive recursion for the communication function, which in turn requires A to be primitive decidable.

Definition 15 (*Primitive*) An aprACP$_R(A,|)$-term is *primitive effective* if
- A is a primitive decidable set,
- $|$ is given as a primitive recursive function $| : A^2 \to A \dot{\cup} \{\delta\}$,
- it contains only primitive recursive guarded recursion,
- only encapsulation operators ∂_H for which H is primitive decidable
- and only renamings ρ_R for R such that $\forall a \in A : (\{b \mid R(a,b)\}$ is finite), and the function which yields for any $a \in A$ this finite set is primitive recursive.

A aprACP$_U(A,|)$-expression is *primitive coeffective* if
- A is a primitive decidable set,
- $|$ satisfies $\forall c \in A : (\{(a,b) \mid a|b = c\}$ is finite), and the total function which yields for any $c \in A$ this finite set is primitive recursive,
- it contains only primitive recursive guarded recursion,
- only encapsulation operators ∂_H for which H is primitive decidable
- and only renamings ρ_R for R such that $\forall b \in A : (\{a \mid R(a,b)\}$ is finite), and the function which yields for any $b \in A$ this finite set is primitive recursive.

A aprACP$_U(A,|)$ expression is *primitive decidable* if it is either primitive effective and without the constant U or primitive coeffective.

The languages of primitive (co)effective aprACP expressions will be denoted aprACP$^{\text{p.e.}}_{R}(A, |)$ and aprACP$^{\text{p.c.}}_{U}(A, |)$. It is easy to see that the signatures and sets of open and closed terms of these languages are decidable. As it is very easy to recognise the sources of primitive recursive functions, they are even primitive decidable. Also the canonicity results of Section 3.7, as well as the expressibility results of Sections 3.5 and 3.6 apply to these variants of aprACP.

The *primitive decidable* process graphs are defined just like the decidable ones (Definition 11), but with the additional requirement that the involved algorithm uses only primitive recursion. Exactly as before it follows that

Proposition 8 Primitive effective process expressions denote only primitive recursive process graphs. Primitive decidable expressions denote only primitive decidable graphs.

Every primitive recursive process graph is denoted by a primitive effective process expression. □

4.7 The Regular and the Recursion-free Expressions

Finally the *regular* process expressions are the ones with finite guarded recursion, and no other operators than inaction, action prefix and choice occurring in recursive specifications, whereas the *recursion-free* ones obviously have no recursion at all. It is easy to show that regular process expressions denote only regular process graphs, and recursion-free expressions only finite and acyclic process graphs. Conversely, every regular process graph is denotable by a regular process expression, and every every finite and acyclic process graph is, up to bisimulation equivalence, denotable by a recursion-free expression.

5 De Simone Languages

A *De Simone language* is a language of which the syntax is given as an annotated signature, and the semantics as a TSS over that signature of a particular form, known as the *De Simone* format.

Definition 16 (*The De Simone format*). A TSS is in the *De Simone* format if for every recursive specification S and $X \in V_S$ it has a rule

$$\frac{S_x[\langle Y|S\rangle/Y]_{Y\in V_S} \xrightarrow{a} z}{\langle X|S\rangle \xrightarrow{a} z}$$

and each of its other rules (the *De Simone rules*) has the form

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

where $(f, n) \in \Sigma$, $I \subseteq \{1, \dots, n\}$ and $t \in \mathbb{T}(\Sigma)$ is univariate recursion-free term containing no other variables than x_i $(1 \leq x \leq n$ and $i \notin I)$ and y_i $(i \in I)$. Here *univariate* means that each variables occurs at most once.
In a rule of the above form, (f, n) is the *type*, a the *action*, t the *target*, and the tuple $(l_1, \dots, l_n)$ with $l_i = a_i$ if $i \in I$ and $l_i = *$ otherwise, the *trigger* [14].

Most process description languages encountered in the literature, including CCS, SCCS, CSP, ACP and MEIJE, are De Simone languages. De Simone languages are known to satisfy all the sanity requirements of Section 2.2 up to bisimulation equivalence. Below I will generalise the classification of process expressions in aprACP$_R$ to a classification of De Simone languages. I will not consider the classes of finite, regular and κ-bounded expressions. The De Simone languages from Definition 16 are the countable ones.

Definition 17 (*Guarded*). Let $P = (\Sigma, A, R)$ be a TSS in De Simone format. For $(f, n) \in \Sigma$ and $1 \leq i \leq n$, the i^{th} argument of (f, n) is *awake* if there is a rule in R of type (f, n) with i in its index set. For $t \in \mathbb{T}^r(\Sigma)$ a free occurrence of a variable in t is *awake* or *unguarded* if for every subterm $f(t_1, ..., t_n)$ of t such that the occurrence is in t_i, the i^{th} argument of f is awake.

Let S be a recursive specification. The relation $\xrightarrow{u} \subseteq V_S \times V_S$ is given by $X \xrightarrow{u} Y$ iff Y is awake in S_X. S is *guarded* if the relation $\xrightarrow{u}$ is well-founded.

This notion of guardedness is due to VAANDRAGER [14]. Guarded recursive specifications in any De Simone language have unique solutions.

Definition 18 A TSS P in De Simone format is said to be

- *(recursively) enumerable* if Σ is decidable, only primitive recursive recursion is allowed, and the set of De Simone rules is r.e.
- *bounded* [14] if only guarded recursion is allowed and for each type and trigger the set of rules involving that type and trigger is finite.
- *effective* [14] if Σ is decidable, only computable guarded recursion is allowed, and there exists a total recursive function associating with each type and trigger the finite set of rules with that type and trigger.
- *coeffective* if Σ is decidable, only computable guarded recursion is allowed, and there exists a total recursive function associating with each type, action and target the finite set of corresponding rules.
- *primitive effective* if Σ is primitive decidable, only primitive recursive guarded recursion is allowed, and there exists a primitive recursive function associating with each type and trigger the finite set of corresponding rules.
- *primitive coeffective* if Σ is primitive decidable, only primitive recursive guarded recursion is used, and there is a primitive recursive function giving for each type, action and target the finite set of corresponding rules.

It is not difficult to apply these requirements to the De Simone language aprACP$_U$ and verify that they coincide exactly with the ones from Section 4.

Proposition 9 Terms in a De Simone language with a property on the left

countable	countable
bounded	finitely branching
recursively enumerable	recursively enumerable
effective	computable
coeffective	decidable
primitive effective	primitive recursive
primitive coeffective	primitive decidable

denote only process graphs satisfying the corresponding property on the right.
Proof: Straightforward. □

The main results announced in this extended abstract concern the expressibility of arbitrary De Simone languages in aprACP$_R$. In order to state which expressiveness results have been obtained, more properties of De Simone languages need to be defined.

Definition 19 (*Dependence of operators*). Let $P = (\Sigma, A, R)$ be a TSS in De Simone format, then *dependence* is the smallest transitive binary relation on Σ such that (f, n) *is dependent on* (g, m) if there is a rule with type (f, n) and with (g, m) occurring in its target.

Definition 20 A TSS P in De Simone format is said to be

- *width-finitary* if for each type there are only finitely many targets (such that there is a rule with that type and target).
- *(primitive) width-effective* if there exists a (primitive) recursive function giving for each type the finite set of corresponding targets.
- *finitary* if
  - (*depth*:) each type is dependent on only finitely many other types,
  - (*width*:) and for each type there are only finitely many targets.
- *image-finite* if for each type and trigger the matching set of rules is finite.
- *functional* if there exists a finite upperbound for the number of rules with any given type and trigger.

The first two properties can best be understood when viewing a De Simone language as an abstract TSS. Width-finitariness is then the property that for every type there are only finitely many abstract rules. (Primitive) width-effectiveness moreover requires that there is a (primitive) recursive function associating with each type the finite set of abstract rules of that type. A language is finitary if the behaviour of a finite term can be deduced by considering only finitely many abstract rules. Thus a finitary De Simone language can be obtained as the combination of a number of De Simone languages with finitely many abstract rules, each of which is trivially primitive width-effective. Boundedness is the combination of guarded- and image-finiteness. As seen in Section 3, aprACP$_R$ is width-finitary, and even primitive width-effective. As in aprACP$_R$ every type is dependent only on itself (at most), the language is finitary as well. Image-finiteness for aprACP$_R$ reduces to image-finiteness of the relational renamings, and functionality corresponds with the restriction to aprACP$_F$.

The following table lists a number of *translatable properties* of De Simone languages. Every translatable property consist of a full row in the table, thus being the conjunction of a left and a right side. For aprACP$_R$, in each property the left side is either always true or implied by the right side.

Theorem 4 Any De Simone language satisfying certain translatable properties of Table 3 is expressible in the version of aprACP$_R$ with the same properties.

Proof: To be supplied in the full version of this paper. My proof is an adaptation of De Simone's construction, but avoids, when possible, the use of unguarded recursion, by resorting to a richer synchronisation algebra $(A, |)$. There is essentially only one construction, translating any open term in any (countable) De Simone language into aprACP$_R$. For any of the properties on the

width-finitary	with guarded recursion
width-effective	with computable guarded recursion
primitive width-effective	with prim. rec. guarded recursion
finitary	with finite guarded recursion
image-finite	with image-finite renaming
functional	with functional renaming
–	recursively enumerable
primitive width-effective	primitive effective

Table 3: Properties of De Simone languages preserved under translation to aprACP$_R$

right in Table 3, I followed the construction backwards to see which additional requirements on De Simone languages are needed to ensure that the version of aprACP they are translated into meets that property. This yielded the properties on the left. □

By taking the dependencies between the properties in Table 3 into account—finite recursion is surely primitive recursive, primitive effectivity entails both image-finiteness and primitive recursive guarded recursion, and with unguarded recursion aprACP$_F$ is as good as aprACP$_R$—Theorem 4 establishes 30 expressibility results. Since virtually all De Simone languages encountered in practice are finitary, the most significant results are

1. Any finitary De Simone language is expressible in aprACP$_R$ with finite guarded recursion.
2. Any finitary image-finite De Simone language is expressible in aprACP$_R$ with finite guarded recursion and image-finite renamings.
3. Any finitary functional De Simone language is expressible in aprACP$_F$ with finite guarded recursion.
4. Any finitary enumerable De Simone language is expressible in aprACP$_R^{\text{r.e.}}$ with finite guarded recursion.
5. Any finitary enumerable image-finite De Simone language is expressible in aprACP$_R^{\text{r.e.}}$ with finite guarded recursion and image-finite renamings.
6. Any finitary enumerable functional De Simone language is expressible in aprACP$_F^{\text{r.e.}}$ with finite guarded recursion.
7. Any finitary primitive effective De Simone language is expressible in aprACP$_R^{\text{p.e.}}$ with finite guarded recursion.
8. Any finitary primitive effective functional De Simone language is expressible in aprACP$_F^{\text{p.e.}}$ with finite guarded recursion.

In each of these results the De Simone languages are also assumed to have finite guarded recursion only, but, by the compositionality of recursion, the same results hold without requiring or getting guardedness, finiteness or both.

Result 4 generalises the original theorem by De Simone, saying that any finitary recursively enumerable De Simone language with finite recursion is expressible in the recursively enumerable version of MEIJE with finite recursion. The generalisation is that, under the assumption that the source languages have only guarded recursion, the target language (now aprACP$_R$) can be required to use only guarded recursion as well.

Using the constant U yields an even stronger result for recursive enumerable De Simone languages, namely by dispensing with finitariness. This result has no effective counterpart.

Theorem 5 Any recursively enumerable De Simone language is expressible in aprACP_U with finite guarded recursion.

Acknowledgments Many thanks to Hanna Walińska and Anna Patterson for proofreading, and to the editors and publisher of this proceeding for delaying publication until my contribution was ready.

References

[1] D. Austry & G. Boudol (1984): *Algèbre de processus et synchronisations.* TCS 30(1), pp. 91–131. See also [4].

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

[3] J.A. Bergstra & J.W. Klop (1984): *The algebra of recursively defined processes and the algebra of regular processes.* In J. Paredaens, editor: *Proc. 11th ICALP, Antwerpen*, LNCS 172, Springer-Verlag, pp. 82–95.

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

[5] S.D. Brookes, C.A.R. Hoare & A.W. Roscoe (1984): *A theory of communicating sequential processes.* JACM 31(3), pp. 560–599.

[6] J.F. Groote & F.W. Vaandrager (1992): *Structured operational semantics and bisimulation as a congruence.* I&C 100(2), pp. 202–260.

[7] Yu.I. Manin (1977): *A Course in Mathematical Logic*, *Graduate Texts in Mathematics* 53. Springer-Verlag.

[8] R. Milner (1980): *A Calculus of Communicating Systems*, LNCS 92. Springer-Verlag.

[9] R. Milner (1983): *Calculi for synchrony and asynchrony.* TCS 25, pp. 267–310.

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

[11] A. Ponse (1992): *Computable processes and bisimulation equivalence.* Report CS-R9207, CWI, Amsterdam.

[12] R. de Simone (1984): *On* Meije *and SCCS: infinite sum operators vs. non-guarded definitions.* TCS 30, pp. 133–138.

[13] R. de Simone (1985): *Higher-level synchronising devices in* Meije–*SCCS.* TCS 37, pp. 245–267. For more details see [12] and: *Calculabilité et Expressivité dans l'Algebra de Processus Parallèles* Meije, Thèse de 3e cycle, Univ. Paris 7, 1984.

[14] F.W. Vaandrager (1993): *Expressiveness results for process algebras.* In J.W. de Bakker, W.P. de Roever & G. Rozenberg, editors: *Proceedings REX Workshop on Semantics: Foundations and Applications,* Beekbergen, The Netherlands, June 1992, LNCS 666, Springer-Verlag, pp. 609–638.