April 1994

Process Algebra, CCS, and Bisimulation Decidability

Seth Kulick

University of Pennsylvania, skulick@cis.upenn.edu

Process Algebra, CCS, and Bisimulation Decidability

Abstract

Over the past fifteen years, there has been intensive study of formal systems that can model concurrency and communication. Two such systems are the Calculus of Communicating Systems, and the Algebra of Communicating Processes. The objective of this paper has two aspects; (1) to study the characteristics and features of these two systems, and (2) to investigate two interesting formal proofs concerning issues of decidability of bisimulation equivalence in these systems. An examination of the processes that generate context-free languages as a trace set shows that their bisimulation equivalence is decidable, in contrast to the undecidability of their trace set equivalence. Recent results have also shown that the bisimulation equivalence problem for processes with a limited amount of concurrency is decidable.

Comments

The Institute For Research In Cognitive Science

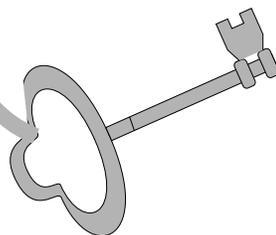**Process Algebra, CCS, and Bisimulation Decidability**

by

Seth Kulick

University of Pennsylvania
3401 Walnut Street, Suite 400C
Philadelphia, PA 19104-6228

April 1994

Site of the NSF Science and Technology Center for

Research in Cognitive Science

P

E

N

N

Process Algebra, CCS, and Bisimulation Decidability *

Seth Kulick

Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

April 20, 1994

Abstract

Over the past fifteen years, there has been intensive study of formal systems that can model concurrency and communication. Two such systems are the Calculus of Communicating Systems, and the Algebra of Communicating Processes. The objective of this paper has two aspects: (1) to study the characteristics and features of these two systems, and (2) to investigate two interesting formal proofs concerning issues of decidability of bisimulation equivalence in these systems. An examination of the processes that generate context-free languages as a trace set shows that their bisimulation equivalence is decidable, in contrast to the undecidability of their trace set equivalence. Recent results have also shown that the bisimulation equivalence problem for processes with a limited amount of concurrency is decidable.

1   Introduction

Over the past fifteen years, there has been intensive study of formal systems that can model concurrency and communication. These formalisms allow both rigorous specification of desired systems and the means to verify the correctness of such specifications. The objective of this paper has two aspects: (1) to study the characteristics and features of two such systems, and (2) to investigate some interesting formal proofs that have arisen in the context of these systems.

One such formalism is the *Calculus of Communicating Systems* (CCS), developed by Robin Milner, and the *Algebra of Communicating Processes* (ACP), developed by a group led by Bergstra and Klop at Amsterdam. Although strongly related, they start from different viewpoints and have different strengths and weaknesses. Collectively they will be referred to in this paper as "process calculi". Section 2 develops CCS but will also be used to explicate the general principles that underly both CCS and ACP. Section 3 describes ACP by comparing and contrasting it with CCS.

One crucial aspect of both these systems is the concept of bisimulation equivalence. Bisimulation decidability for the full CCS and ACP systems is undecidable, but for smaller subsets of the system some surprising results have been found. In particular, in a subset of ACP it has been shown that the decidability of bisimulation equivalence for processes generating context-free languages is decidable, in contrast to the well-known undecidability of context-free languages in automata theory. Another

---
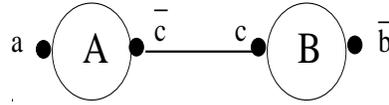
Figure 1: The composite agent $A \mid B$

decidability result, within the context of CCS, allows for a limited amount of concurrency. I will give an overview of these decidability results and sketch the methods of proof.

2   CCS

2.1   Definition and Features

The goal of the Calculus of Communicating Systems [Milner 1989] is to formalize a theory of concurrent processing in terms of a few primitive notions, in which communication is the central primitive. Each process in the system is called an *agent*, and the agents communicate with each other in a limited way via their input/output *ports*. Arbitrary agents are referred to by an element from the set of agents $\mathcal{P} = \{P, Q, \ldots\}$, and agents that are explicitly defined are denoted by an element from the set of *agent constants* $\mathcal{K} = \{A, B, \ldots\}$ [1] An agent's input ports are specified by the set of *names* $\mathcal{A} = \{a, b, c, \ldots\}$ and the output ports are specified by the set of *co-names* $\overline{\mathcal{A}} = \{\overline{a}, \overline{b}, \ldots\}$. Any output port $\overline{x}$ may only communicate with its corresponding input port x. This enforces the underlying idea of CCS that handshaking is the essential communication primitive. However, it need not be a one-to-one relationship; e.g., there can be many input ports on different agents all connected to the the same output port. An action is (almost) therefore the same as specifying the name of a port. The one exception is the *silent action* τ, to be explained shortly. The set of labels $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$ and the set of *actions* $\mathsf{Act} = \mathcal{L} \cup \{\tau\}$.

A simple example of an agent definition is that for a one-element buffer [2] ,with input port a and output port $\overline{b}$:

$$
\begin{aligned}
C &\stackrel{def}{=} a.C' \\
C' &\stackrel{def}{=} \overline{b}.C
\end{aligned}
$$

Now consider a buffer of length two created by hooking two copies of the agent C together (call them C_0 and C_1). [3] A problem is that the output $\overline{b}$ of C_0 will be hooked to the input a of C_1, which violates the condition on port interaction. Therefore, an unused port name, e.g. c [4] , is used to *relabel* C_0's output port to be $\overline{c}$ and B's input port to c. The *composition* of C_0 and C_1, written $C_0 \mid C_1$, can now take place. For simplicity, let A and B now refer to C_0 to C_1, and so the composition appears as in Figure 1. However, this is still not quite right, because we need to enforce the handshaking communication - that is, when C_0 sends out on $\overline{c}$, that it is received and acted upon by B. Also, this is purely internal and so should be "hidden" from agents outside of $A \mid B$. Both these aims are achieved by imposing the *restriction* $\backslash c$ upon $A \mid B$, which signifies that the restricted composite agent $(A \mid B) \backslash c$ may not perform c or $\overline{c}$ actions, although it may, crucially, perform a *silent step*, written τ, which results from the communication $(c, \overline{c})$ between C_0

---

[1] These name restrictions will be relaxed in examples for clarity.

[2] For simplicity, the actual values passed through the buffer will be ignored for now

[3] There is no connection between the names of the agents and the names of the ports

[4] Really, anything except a or b.
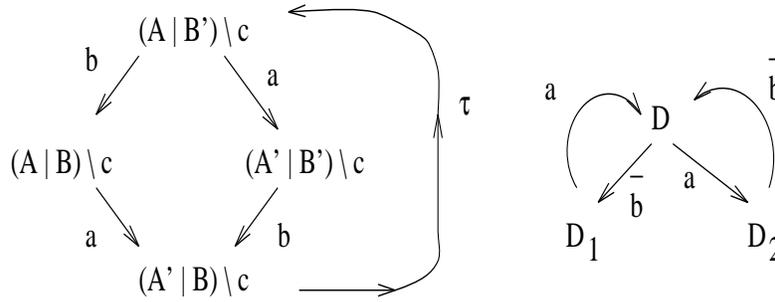
Figure 2: Derivation trees in CCS

and B. In terms of transition rules, we can have $(A \mid B) \xrightarrow{a} (A' \mid B)$ but not $(A' \mid B) \xrightarrow{\bar{c}} (A \mid B)$. What is allowed instead is $(A' \mid B) \xrightarrow{\tau} (A \mid B')$.

A formal summary of the discussion above is provided the definition of the set $\mathcal{E}$ of *agent expressions*, which defines the possible ways in which agents can be joined together via communication. $\mathcal{E}$ is the smallest set which includes $\mathcal{K}$ and the following expressions, where E and E_i are already in $\mathcal{E}$:

$$
\begin{array}{ll}
\alpha.E & \text{prefix } (\alpha \in \mathsf{Act}) \\
\sum_{i \in I} & \text{summation } (I \text{ an indexing set}) \\
E_1 \mid E_2 & \text{composition} \\
E \setminus L & \text{restriction}(L \subset \mathcal{L} \\
E[f] & \text{relabelling } (f \text{ a relabelling function})
\end{array}
$$

2.1.1   Recursive Equations

Processes may sometimes be defined as the solution of a finite system of recursive equations $\{X_i \overset{def}{=} E_i\}$. In order to guarantee a unique solution to such a system, it is usually required that the E_i be *guarded*, meaning that each variable X_i in E_i is preceded by an atomic action. For an extreme example of where this is not true, consider the equation $X = X$, which is not guarded, and for which every process is a solution. The recursive equations in this paper will all be guarded. [5]

2.1.2   derivation trees and bisimulation

Any agent will have a certain set of possible transitions, which can be collected into a *derivation tree*. For example, the left side of Figure 2 shows the derivation tree for $(A \mid B)\backslash c$. The comparison of derivation trees for different processes is a crucial aspect of CCS (and process calculi general), and CCS has three different concepts for this purposes, which equate trees based on varying notions of "similiarity". This is the crucial departing point between CCS and classical automata theory. Whereas in the latter the resulting language is the object of concern, here we are concerned with the branching activity of the process, as represented in the derivation tree or process definition.

The first notion of "similarity" is that of *strong bisimulation*, in which every α action of one agent must be matched by an α action of the other, even for τ actions. A pair of examples before the formal definition:

---

[5]The theory of guarded equations and unique solutions is actually considerably more complex and subtle than described here, but the extra complexity will not be a factor in the examples in this paper. The brief discussion above is just meant to introduce the notion of guarded equations, which will be needed later.

1. $A = a(b + c), B = ab + ac$. These two are *not* strongly bisimilar. Note that this entails a rejection of the distributive law. In A, *first* a must be executed, and *then* a choice is made between b and c. However, in B, *first* a choice is made, and *then* the chosen term is executed. The moment of choice is different, and so they are distinguished by strong bisimulation.

2. $A' = ab, B' = ab + a(b + b)$. These two *are* strongly bisimilar. Based only on what action is taking place, A' and B' cannot be distinguished.

Note that the last example shows that it would be too strong to require identical derivation trees.

Definition 1 *A binary relation $S \subseteq P \times P$ over agents is a* strong bisimulation *if $(P, Q) \in S$ implies, for all $\alpha \in$ Act,*
(i) Whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in S$
(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then, for some P', $P \xrightarrow{\alpha} P'$ and $(P', Q') \in S$.

Definition 2 *P and Q are* strongly bisimilar, *written $P \sim Q$, if $(P', Q') \in S$ for some strong bisimulation S. Equivalently, $\sim$ is the largest strong bisimulation or:*

$$\sim = \cup\{S \mid S \text{ is a strong bisimulation}\}$$

Strong bisimulation is the simplest of the three equivalence definitions because it treats τ just like any other action. Recall that τ is supposed to represent an action that is "hidden" from observance, and thus it should not really be treated just like any other action. The various ways of handling the silent action leads to a variety of ways for comparing derivation trees. One of the most important variations is the requirement that each τ action be matched by zero or more τ actions ; this is called *weak bisimulation*. For example, in Figure 2 the two derivation trees are not strongly bisimilar, but are indeed weakly bisimilar, because S is a bisimulation, where

$$
\begin{aligned}
S \quad = \quad & \{((A \mid B') \setminus c, D) \\
& ((A \mid B) \setminus c, D_1), \\
& ((A' \mid B') \setminus c, D_2), \\
& ((A' \mid B) \setminus c, D)\}
\end{aligned}
$$

Two preliminary definitions are needed before the definition of weak bisimulation:

1. **Definition 3** *if $t \in$ Act*, then $\hat{t} \in \mathcal{L}^*$ is the sequence obtained by deleting all occurrences of τ from t. In particular, $\widehat{\tau^n} = \varepsilon$.*

2. **Definition 4** *If $t = \alpha_1 \ldots \alpha_n \in$ Act*, then $E \xRightarrow{t} E'$ if*

$$E(\xrightarrow{\tau})^*(\xrightarrow{\alpha_1})(\xrightarrow{\tau})^* \ldots (\xrightarrow{\tau})^*(\xrightarrow{\alpha_n})(\xrightarrow{\tau})^* E'$$

    Note also that $E \xRightarrow{\varepsilon} E'$ iff $E \xrightarrow{\tau^n} E'$ for some $n \geq 0$.

Definition 5 *A binary relation $S \subseteq P \times P$ over agents is a* (weak) bisimulation *if $(P, Q) \in S$ implies, for all $\alpha \in$ Act,*
(i) Whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \xRightarrow{\hat{\alpha}} Q'$ and $(P', Q') \in S$
(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then, for some P', $P \xRightarrow{\hat{\alpha}} P'$ and $(P', Q') \in S$.

Definition 6 *P and Q are* (weakly) bisimilar, *written $P \approx Q$, if $(P', Q') \in S$ for some (weak) bisimulation S. Equivalently, $\approx$ is the largest weak bisimulation or:*

$$\sim = \cup\{S \mid S \text{ is a weak bisimulation}\}$$

A crucial property of bisimulation is that for any agent P, $P \approx \tau.P$ (this is not true for $\sim$). This is exactly what allows τ to be ignored to a certain extent when comparing agents. Consider, however, $P = a.0 + b.0$ and $Q = a.0 + \tau.b.0$. P is deterministic in the sense that there is *always* available a choice between a and b, while in Q, because of the silent action, the choice for a may no longer be available even though to the external observer there appears to have been no action. Therefore, even though $b.0 \approx \tau.b.0$, $a.0 + b.0 \not\approx a.0 + \tau.b.0$. Thus, $\approx$ is not a congruence relation with respect to summation.

The final notion of equivalence is aimed at capturing the largest congruence relation included in $\approx$:

Definition 7 *P and Q are* equal *or* observation-congruent, *written $P = Q$, if for all α,*
(i) Whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \xRightarrow{\alpha} Q'$ and $P' \approx Q'$
(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then, for some P', $P \xRightarrow{\alpha} P'$ and $P' \approx Q'$

One important law of equality is that $\alpha.\tau.P = \alpha.P$. Note that equality lies between strong and weak bisimilarity: $P \sim Q$ implies $P = Q$ implies $P \approx Q$.

2.1.3   Expansion law

The expression for the two-cell buffer, $(C_0 \mid C_1) \setminus c$, is typical of many system expressions, and so a restricted composition of relabelled components is called *standard concurrent form (scf)*. Its general format is:

$$(P_1[f_1] \mid \ldots \mid P_n[f_n]) \setminus L.$$

One more important law of CCS is the *expansion law*, which is concerned with the immediate actions of an agent in standard concurrent form. These actions could result from two possibilities:

1. the action α of a single component P_i. Then the scf will have an action $f_i(\alpha)$, and result in the new scf $(P_1[f_1] \mid \ldots \mid P_i'[f_i] \mid \ldots \mid P_n[f_n]) \setminus L$ meaning that only the ith component has changed.

2. a τ action, a communication resulting from actions l_i and l_j by P_i and P_j, respectively (for $1 \leq i < j \leq n$), such that $f_i(l_i) = \overline{f_j(l_j)}$. The result is the new scf $(P_1[f_1] \mid \ldots \mid P_i'[f_i] \mid \ldots \mid P_j'[f_j] \mid \ldots \mid P_n[f_n]) \setminus L$ meaning that exactly two components have changed.

Proposition 8 *The Expansion Law* [6] *Let $P \equiv (P_1 \mid \ldots \mid P_n) \setminus L$ with $n \geq 1$. Then*

$$
\begin{aligned}
P &= \sum\{\alpha.(P_1 \mid \ldots \mid P_i' \mid \ldots \mid P_n) \setminus L : P_i \xrightarrow{\alpha} P_i', \alpha \notin \mathcal{L} \cup \overline{\mathcal{L}}\} \\
&+ \sum\{\tau.(P_1 \mid \ldots \mid P_i' \mid \ldots \mid P_j' \mid \ldots \mid P_n) \setminus L : P_i \xrightarrow{l} P_i', P_j \xrightarrow{\bar{l}} P_j', i < j\}
\end{aligned}
$$

Essentially, repeated application of the expansion law will result in the derivation tree for a process. The expansion law will be a crucial aspect of the decidability proof in section (4.2).

---

[6]This is actually a simplified version (without any renaming functions allowed) used for clarity. With renaming, it would work along the lines indicated by the immediately previous discussion.
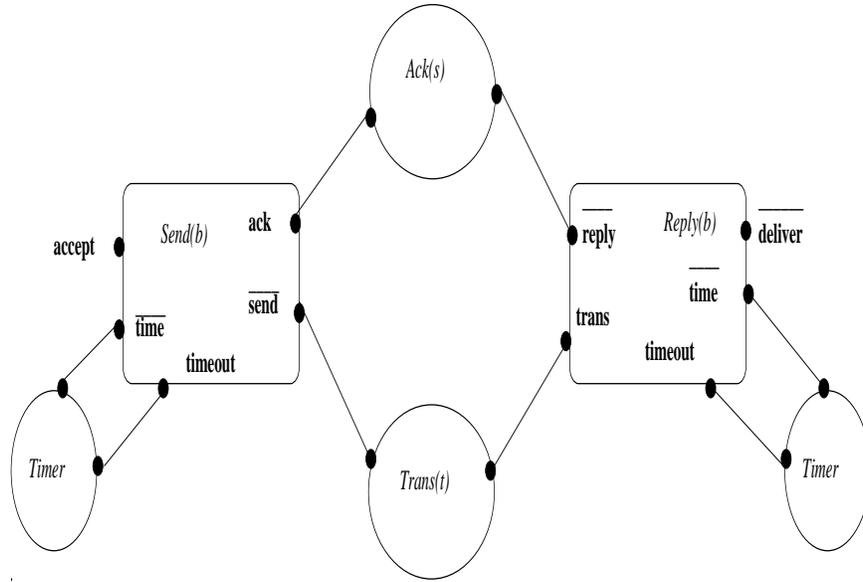
Figure 3: Alternating bit protocol in CCS

2.2   Verifying the buffer

The previous section defined a buffer of length two created by linking two single-element buffers together. It is desired to verify that the two buffers together work as if they were actually one two-element buffer. In order to verify this, we first need an independent specification of a two-element buffer and then we need to prove that the two specifications specify equivalent processes. Once again, C_0 and C_1 will be referred to as A and B, respectively. Now consider a specification for a buffer of length 2, $Buf_2(0)$:

$$
\begin{aligned}
Buf_2(0) &\stackrel{def}{=} a.Buf_2(1) \\
Buf_2(1) &\stackrel{def}{=} a.Buf_2(2) + \bar{b}.Buf_2(0) \\
Buf_2(2) &\stackrel{def}{=} \bar{b}.Buf_2(1)
\end{aligned}
$$

Thus the goal is to prove that $(A \mid B) \setminus c \approx Buf_2(0)$. Note, however, that it was already shown that $(A \mid B) \setminus c \approx D_1$, where $D_1 = a.D$ and $D = a.\bar{b}.D + \bar{b}.a.D$. Thus, D and $Buf_2(1)$ denote exactly the same process, and the same bisimulation is used to show that $(A \mid B) \setminus c \approx Buf_2(0)$.

2.3   Specifying and Verifying the Alternating Bit Protocol

The alternating bit protocol (hereafter ABP), is a protocol designed to ensure reliable communication through unreliable transmission mediums (see Figure 3). Here, $Send$ and $Reply$ will be agents that accept and deliver messages. $Send$ routes messages through a medium represented by the process $Trans$, and $Reply$ responds to $Send$ through a medium represented by the agent Ack. Messages get tagged with bits 0 and 1, alternately. After $Sender$ gets a message, it sends it with a bit b along the $Trans$ line and sets a timer. There are then three possibilities:

- it may get a 'time-out' from a timer, upon which it sends the message again with b.

- it gets an acknowledgment b from Ack, which means that the message made it through, and thus is now ready to accept another message, which it will send with bit $\hat{b} = 1 - b$.

- it gets an acknowledgment $\hat{b}$, which it ignores.

After the replier delivers a message, it acknowledges it with a bit b along the Ack line and sets a timer. There are also three possibilities:

- it gets a 'time-out' from the timer, upon which it sends its acknowledgment b again.

- it gets a new message with bit $\hat{b}$ from $Trans$, which it then delivers, and acknowledges with bit $\hat{b}$.

- it gets a transmission of the previous message with bit b, which is ignored.

If $Send(b)$ and $Reply(b)$ are composed with their timers, under the restriction {**time,timout** }, then the definitions of $Send(b)$ and $Reply(b)$ are:

$$
\begin{aligned}
Send(b) &= \overline{\mathbf{send_b}}.Sending(b) \\
Sending(b) &= \tau.Send(b) + \mathbf{ack_b}.Accept(\hat{b}) + \mathbf{ack_{\hat{b}}}.Sending(b) \\
Accept(b) &= \mathbf{accept}.Send(b) \\
Reply(b) &= \overline{\mathbf{reply_b}}.Replying(b) \\
Replying(b) &= \tau.Reply(b) + \mathbf{trans_{\hat{b}}}.Deliver(\hat{b}) + \mathbf{trans_b}.Replying(b) \\
Deliver(b) &= \overline{\mathbf{deliver}}.Reply(b)
\end{aligned}
$$

Note that the silent action represents the internal communication between a timer and the sender or receiver. The communication lines $Trans$ and Ack will be defined by giving transition equations. It is assumed that these lines may lose or duplicate, but not corrupt messages, and that buffers have an unbounded message capacity.

$$
\begin{aligned}
Ack(bs) &\overset{\overline{\mathbf{ack_b}}}{\rightarrow} Ack(s) & Trans(sb) &\overset{\overline{\mathbf{trans_b}}}{\rightarrow} Trans(s) \\
Acks(s) &\overset{\mathbf{reply_b}}{\rightarrow} Ack(sb) & Trans(s) &\overset{\mathbf{send_b}}{\rightarrow} Trans(bs) \\
Ack(sbt) &\overset{\tau}{\rightarrow} Ack(st) & Trans(tbs) &\overset{\tau}{\rightarrow} Trans(ts) \\
Ack(sbt) &\overset{\tau}{\rightarrow} Ack(sbbt) & Trans(tbs) &\overset{\tau}{\rightarrow} Trans(tbbs)
\end{aligned}
$$

Note that sbt represents the concatenation of s, b, t, where $s, t \in \{0, 1\}^*, b \in \{0, 1\}$. The last two lines of Ack and $Trans$ represent loss and duplication, respectively, of any bit in transit.

So to represent the complete system, let

$$
AB \overset{def}{=} (Accept(\hat{b}) \mid Trans(\varepsilon) \mid Ack(\varepsilon) \mid Reply(b)) \setminus L
$$

where L is the set of all internal actions (that is, all actions except **accept** and **deliver**. AB represents the state in which a message has just been delivered, a new message is about to be accepted, and the transmission lines are empty. AB is the definition of the protocol. It specification is that it should act as a simple buffer, as follows:

$$
\begin{aligned}
Buf &\overset{def}{=} \mathbf{accept}.Buf' \\
Buf' &\overset{def}{=} \overline{\mathbf{deliver}}.Buf
\end{aligned}
$$

and so to prove that ABP meets its specification, it needs to be shown that $ABP \approx Buf$. Such a bisimulation S can be found:

AB States	Buf states
$Accept(\hat{b}) \mid Trans(b^n) \mid Ack(B^p) \mid (Reply(b) \text{ or } Replying(b))$	Buf
$(Send(\hat{b}) \text{ or } Sending(\hat{b})) \mid Trans(\hat{b}^m b^n) \mid Ack(b^p) \mid (Reply(b) \text{ or } Replying(b))$	Buf'
$(Send(\hat{b}) \text{ or } Sending(\hat{b})) \mid Trans(\hat{b}^m) \mid Ack(b^p) \mid Deliver(\hat{b})$	Buf'
$(Send(\hat{b}) \text{ or } Sending(\hat{b})) \mid Trans(\hat{b}^m) \mid Ack(b^p \hat{b}^q) \mid (Reply(b) \text{ or } Replying(b))$	Buf

Note that $b \in \{0, 1\}$ and $m, n, p, q \geq 0$, to represent the arbitrary bit-sequences in the transmission lines. By choosing either of two alternatives where possible for the AB states, there are twelve groups altogether.

Two remarks:

1. A rather tedious case analysis can be used to verify that this is indeed a bisimulation. This type of analysis could be automated, and indeed the search for a bisimulation is an obvious candidate for automation.

2. Nothing disallows the possibility that one of the transmission lines could lose data indefinitely. It is assumed that the behavior of the agents will be "fair" and thus this will not happen. This issue will reappear in the context of ACP.

3 Algebra of Communicating Processes

The other system of process calculus under discussion here is the Algebra of Communicating Processes (ACP). ACP is very closely related to CCS, but with some differences in definition and expressibility, and based on a different methodological approach.

CCS can be seen as fixing a model (the derivation trees of various agents), and deriving various laws based on that model. In constrast, ACP is an axiomatic approach, in which various axioms are stated, and its concern is with *any* model that satisfies those axioms. For example, a model could be one that contains only "finitely branching" [7] processes, or one that allows infinite branching. An advantage of this approach is that it allows, more so than with CCS, an explicit modularization of the various problems and features involved in these systems. ACP is actually built up from a series of smaller systems, and the ideal is that an applications designer could pick just the right axiom set for the desired system. Nothing really prevents such an approach with CCS - indeed, the proof in section (4.2.1) does just that, by choosing a subset of CCS. The designers of ACP, however, has gone much further with such an approach. This paper will follow Baeten & Weijland in [Baeten and Weijland 1990] and refer to all of the various axiomatic systems that lead up to ACP *Process Algebras*. [8] The first such system that will be examined is called *Basic Process Algebra*

---

[7] A finitely branching graph has only finitely many edges leaving from each node

[8] This is a somewhat unfortunate choice of names, since one of the axiomatic systems is itself called "Process Algebra". However, the context should make clear what is being referred to.

(BPA), and will be gone into in some detail because it is the system used for the proof in section (4.1).

3.1   Basic Process Algebra

The signature of BPA [9] consists of the set of atomic actions $\mathsf{A} = \{a, b, \ldots\}$, a set of variables $\{x, y, \ldots\}$, and the binary operators $+$ and $\cdot$ [10] BPA consists of its signature together with the following axiom set:

$$
\begin{array}{ll}
x + y = y + x & A1 \\
(x + y) + z = x + (y + z) & A2 \\
x + x = x & A3 \\
(x + y)z = xz + yz & A4 \\
(xy)z = x(yz) & A5
\end{array}
$$

If some M is a model for BPA, then the elements of its domain are called *processes*. The variables in the axiom equations stand for processes for some arbitrary model of BPA, and are assumed to be universally quantified. Some remarks on the BPA axioms:

1. BPA is a very simple axiom set, as it doesn't even handle concurrency.

2. The semantic meanings of the axioms are the obvious ones. $\cdot$ is *sequential composition*; $x \cdot y$ is the process that first executes x and upon completion of x begins executing y. $+$ is the *alternative composition*; $x + y$ is the process that either executes x, or executes y, but not both.

3. The left-distributive law is *not* included, for the same reason that it was not valid for CCS.

4. One difference between BPA and CCS is already apparent; whereas CCS allowed prefix multiplication (atomic action a and process p can yield $a \cdot p$), BPA allows general multiplication (processes p and q yield $p \cdot q$). Bergstra and Klop [Bergstra & Klop 1985] claim that there exist examples of recursively defined processes that have finite recursive definitions in terms of general, but not prefix, multiplication.

The introduction of general multiplication requires that BPA make more explicit the possibility of deadlock. In CCS, the idea of deadlock was always, in a sense, "lurking in the background" of the idea of looking at branching structure instead of just the traces. With general multiplication in BPA, however, it must be dealt with in a more explicit manner. Consider a process $x \cdot y$, where x is a process that might reach a state of deadlock (for example x might consist of several other processes running in parallel). [11] If x reaches a state of deadlock, then y cannot begin to execute. To describe this possibility, the special constant δ is used to signify deadlock, and the following two axioms are added to A1-5:

$$
\begin{array}{ll}
x + \delta = x & A6 \\
\delta x = \delta & A7
\end{array}
$$

---

[9] The set of constant and function symbols that may appear in the specification

[10] Usually left out, so that $x \cdot y = xy$.

[11] Not strictly expressible in BPA, which can't express concurrency, but this example is just meant to motivate the deadlock constant which is used throughout the entire range of process algebras.

BPA together with A6 and A7 is referred to as BPA_δ. A6 states that as long as there is any alternative that can proceed, there is no deadlock, and A7 states that no other action can follow a deadlock. BPA_δ can also get extended with a counterpart to δ: the new constant ϵ is used to represent an empty process, one that does nothing but have immediate successful termination:

$$x\epsilon = x \quad A8$$
$$\epsilon x = x \quad A9$$

So BPA_ϵ has axioms A1-5 and A8,A9, while $BPA_{\delta\epsilon}$, with both new constants included, includes axioms A1-9. In contrast to CCS, which has one constant to represent termination, BPA has constants for both successful and unsuccessful termination. However, the inclusion of ε significantly complicates the axiom system when concurrency and communication are introduced, and since the examples to be presented to do not require ε, it will only be included in the BPA system.

3.1.1 Some Models for $BPA_{\delta\epsilon}$

A CCS derivation tree corresponds to a *process graph* in the context of BPA:

Definition 9 *A* process graph *is a graph in which every edge has a label from* A*, and in which the nodes may carry a label* $\downarrow$*, which indicates whether or not the state represented by the node has a termination option.* [12]

Using process graphs, a hierarchy of some models for BPA_δ will now be presented. The first model,$\mathbf{G}_\infty$, consists of countably branching process graphs with edge labels from A. Bisimulation is defined in terms of these graphs:

Definition 10 *Let* $g, h \in \mathbf{G}_\infty$ *and let R be a relation between the nodes of g and the nodes of h. R is a* bisimulation *between g and h, written* $R{:}g\underline{\leftrightarrow}h$*, when the same conditions as for strong bisimulation in CCS are satisfied, plus the condition that if R(s,t), then* $s \downarrow$ *iff* $t \downarrow$*.*

$\mathbf{G}_\infty/\underline{\leftrightarrow}$ will be the set of processes that form a model for $BPA_{\delta\epsilon}$. However, in order to be a model, the operators $+$ and $\cdot$ need to be given a meaning in terms of members of $\mathbf{G}_\infty$. Before this can be done, the preliminary notion of *root unwinding* needs to be mentioned: for any process graph $g \in \mathbf{G}_\infty$, its unwound version $\rho(g)$ can be constructed such that $\rho(g)$ has no edges going back to its root, and such that $g\underline{\leftrightarrow}\rho(g)$. This is a simple idea borrowed from basic automata theory.

Definition 11 *For process graphs* $g, h \in \mathbf{G}_\infty$*,* $g + h$ *is constructed by identifying the roots of* $\rho(g), \rho(h)$*.* $g \cdot h$ *is constructed by identifying every node in g having label* $\downarrow$ *with the root node of a distinct copy of* $\rho(h)$*. Every node emerging from such identification has label* $\downarrow$ *iff* $h \downarrow$*. If g has no labels* $\downarrow$*, then the result is just g. (see Figure 4.*

It can be shown that $\mathbf{G}_\infty/\underline{\leftrightarrow} \models BPA_\delta$. Other models for BPA_δ can be obtained by taking smaller subsets of the carrier set with the obvious restrictions on the $+$ and $\cdot$:

1. $\mathbf{G}/\underline{\leftrightarrow}$ - finitely branching process graphs.

---

[12][Baeten and Weijland 1990] are unclear on the meaning of $\downarrow$, but its purpose appears to be that if a node has deadlock, then it will be a leaf without $\downarrow$.
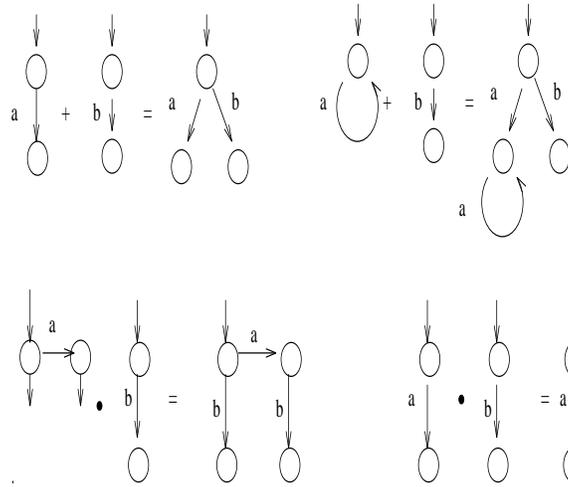
Figure 4: Examples of $+$ and $\cdot$ in BPA

2. $\mathbf{R}/\underline{\leftrightarrow}$ - finite process graphs.

3. $\mathbf{F}/\underline{\leftrightarrow}$ - finite acyclic process graphs.

It's clear the $\mathbf{F} \subset \mathbf{R} \subset \mathbf{G} \subset \mathbf{G}_\infty$. Also, the graphs sets of each of the models can be easily restricted such that they become models for BPA. In particular, $\mathbf{G}/\underline{\leftrightarrow}$ is the model used throughout the decidability proof in section (4.1).

3.2   Process Algebra(PA)

PA is an extension to BPA that can describe processes that are executing in parallel. This is done by introducing two new operators:

1. $\parallel$ - composition(merge) - this is an interleaving composition, as in CCS.

2. $\underline{\parallel}$ - left-merge - this was not in CCS. It helps to simplify various calculations and, according to Baeten & Weijland, it has been proven that the merge operator cannot be finitely axiomatized without such an auxiliary operator.

and the following new axioms:

$$
\begin{array}{ll}
x \parallel y = x \underline{\parallel} y + y \underline{\parallel} x & M1 \\
a \underline{\parallel} x = ax & M2 \\
ax \underline{\parallel} y = a(x \parallel y) & M3 \\
(x + y) \underline{\parallel} z = x \underline{\parallel} z + y \underline{\parallel} z & M4
\end{array}
$$

(a is an arbitrary element of A, and axioms M2 and M3 are in fact axiom schemas, since there is an axiom for each element of A, which is assumed to be finite.) The system PA consists of BPA+M1-M4. Axiom M1 defines the merge in terms of the left-merge: when processes x and y get merged, then either the first step will come from x, or the first step will come y, and axioms M2-M4 define left-merge. [13] Also, δ can be easily included in PA: $PA_\delta = PA + A6, A7$ with the stipulation that in axiom schemas M2,M3, the constant a ranges over $\mathsf{A} \cup \{\delta\}$ (instead of just A).

---

[13] $\cdot$ binds stronger than $\parallel$ or $\underline{\parallel}$, so the left-hand side of M3, for example, stands for $(a \cdot x) \underline{\parallel} y$.

3.2.1 Some models for PA_δ

The model $\mathbf{G}_\infty/\underline{\leftrightarrow}$ can be extended to become a model for PA_δ by defining the operators $\parallel$ and $\underline{\parallel}$.

Definition 12 *For process graphs $g, h \in \mathbf{G}_\infty$, the graph $g \parallel h$ is the cartesian product of the graphs g and h. More precisely:*

1. *the nodes of $g \parallel h$ are all pairs of nodes from g and nodes from h.*

2. *a node (s, t) in $g \parallel h$ has label $\downarrow$ iff both s and t do.*

3. *there is an edge $(s, t) \xrightarrow{a} (s't)$ in $g \parallel h$ precisely if there is an edge $s \xrightarrow{a} s'$ in g; there is an edge $(s, t) \xrightarrow{a} (s, t')$ in $g \parallel h$ precisely if there is an edge $t \xrightarrow{a} t'$ in h*

4. *the root node of $g \parallel h$ is the pair of roots from g and h.*

The graph $g \underline{\parallel} h$ can be constructed as follows:

1. construct $g \parallel h$ and unwind it, getting $\rho(g \parallel h)$.

2. if (s, t) is the root of $\rho(g \parallel h)$, then remove all edges $(s, t) \xrightarrow{a} (s, t')$ where $t \xrightarrow{a} t'$ is an edge in h (that is, remove the edges that originate from h).

3. remove all parts of the graph that have become inaccessible from the root node.

It can be shown that bisimulation is a congruence relation with respect to $\parallel$ and $\underline{\parallel}$, and that $\mathbf{G}_\infty/\underline{\leftrightarrow} \models PA_\delta$. As in the last section, $\mathbf{G}_\infty/\underline{\leftrightarrow}$ can be restricted to form the smaller models.

3.3 ACP

Although the axioms of PA can now handle concurrent processing, there is no method to describe communication between processes. The next extension of the theory, the Algebra of Communicating Processes (ACP), aims to correct this defect.

There are several components to the implementation of communication:

A communication function γ - which is a partial binary function on A. For example, if $\gamma(b, c) = a$, then a is a communication action resulting from b and c, and if $\gamma(b, c)$ is not defined, then b and c do not communicate.

the communication merge operator $|$ - a binary operator on processes. $x \mid y$ represents a merge of two processes x and y with the restriction that the first step is a communication between x and y. In case communication is not defined between the first actions of x and y, then the communication merge is equal to δ. This means that ACP must be extension of PA_δ, not just PA, because the deadlock constant is essential once communication is introduced.

An encapsulation operator ∂_H - For some set of actions $H \subset \mathsf{A}$, ∂_H is a function that renames all members of H to δ, and is otherwise the identity function. Its purpose is to encapsulate a process p w.r.t. H, so that $\partial_H(p)$ cannot communicate with its environment via communication actions in H. Encapsulation in ACP is very close to CCS's restriction operator, but is

$x + y = y + x$	$A1$	$\partial_H(a) = a$ if $a \notin H$	$D1$
$(x + y) + z = x + (y + z)$	$A2$	$\partial_H(a) = \delta$ if $a \in H$	$D2$
$x + x = x$	$A3$	$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	$D3$
$(x + y)z = xz + yz$	$A4$	$\partial_H(xy) = \partial_H(x) \cdot \partial_H(y)$	$D4$
$(xy)z = x(yz)$	$A5$	$a \mid b = \gamma(a, b)$ if γ defined	$CF1$
$x + \delta = x$	$A6$	$a \mid b = \delta$ otherwise	$CF2$
$\delta x = \delta$	$A7$	$ax \mid b = (a \mid b) \cdot x$	$CM5$
$x \parallel y = x \rotatebox{0}{$\parallel$} y + y \rotatebox{0}{$\parallel$} x + x \mid y$	$CM1$	$a \mid bx = (a \mid b) \cdot x$	$CM6$
$a \rotatebox{0}{$\parallel$} x = ax$	$CM2$	$ax \mid by = (a \mid b) \cdot (x \parallel y)$	$CM7$
$ax \rotatebox{0}{$\parallel$} y = a(x \parallel y)$	$CM3$	$(x + y) \mid z = x \mid z + y \mid z$	$CM8$
$(x + y) \rotatebox{0}{$\parallel$} z = x \rotatebox{0}{$\parallel$} z + y \rotatebox{0}{$\parallel$} z$	$CM4$	$x \mid (y + z) = x \mid y + x \mid z$	$CM9$

Figure 5: ACP axiom set

not quite the same. Unlike the latter, ∂_H does not hide the internal actions from outside processes. That is, it does enforce the restriction that the actions cannot actually affect external actions, but the encapsulated actions can still be seen. When communication is introduced to ACP in section (3.4), the consequence will be that the abstraction act is separated from the communication. Communication will not immediately result in a silent action, but rather yield an internal action that is still visible and will then have to be abstracted away by an abstraction operator.

The axiom set for ACP is show in Figure 5. Note that axiom $CM1$ expands axiom $M1$ of PA to include the possible communication, so that for a merge $x \parallel y$, it can either start with a first step from x ($x \rotatebox{0}{$\parallel$} y$), a first step from y ($y \rotatebox{0}{$\parallel$} x$), or a communication step between x and y ($x \mid y$). Axioms $CM2 - 4$ are the same as $M2 - M4$ of PA. Axioms $CM6 - 9$ define the communication operator.

Note that although the operator $\mid$ in CCS can be compared with $\parallel$ in ACP, there are several differences:

1. the definition of $\parallel$ in CCS uses auxiliary operators ($\rotatebox{0}{$\parallel$}$ and $\mid$) that are not used in CCS.

2. communication is more flexible in ACP than in CCS. Whereas in CCS communication is limited to interaction between co-named ports, in ACP communication is defined by the γ function, and need not even be handshaking communication (i.e., $\gamma(a, b, c) = d$ means that a, b, c communicate together to result in d). It's questionable as to how useful this extra capability is, though. Enforcing handshaking in CCS would simply mean that $\forall a, b, c \in$ A, $\gamma(a, b, c)$ is undefined.

3. encapsulation is ACP is likewise more flexible than restriction in CCS, due to the more flexible communication possibilities. For some process x and action a in CCS, then the CCS expression $x \setminus a$ is equivalent to $\partial_H(x)$ for $H = \{a, \overline{a}\}$ in ACP.

3.3.1   Some models for ACP

The model $\mathbf{G}_\infty / \underline{\leftrightarrow}$ from section (3.2.1) can also be extended to become a model for ACP by defining the operators $\parallel, \rotatebox{0}{$\parallel$}$, $\mid$, and ∂_H There is little point in going through the entire formal

definition, but the basic idea is that for some graphs $g, h \in \mathbf{G}_\infty$, the graph $g \parallel h$ gets constructed basically the same as before except that if for some nodes $r, r' \in g$ and $s, s' \in h$, with $r \xrightarrow{a} r'$ in g and $s \xrightarrow{b} s'$ in h and $\gamma(a, b) = c$ then $g \parallel h$ also has an edge $(r, s) \xrightarrow{c} (r', s')$. Also, $\partial_H(g)$ is obtained from the graph for g by removing all edges with labels from H.

It can be shown that bisimulation is a congruence relation with respect to $\parallel, \parallel\!\!\!\perp, |$, and ∂_H and that $\mathbf{G}_\infty/\underline{\leftrightarrow} \models ACP$. As before, $\mathbf{G}_\infty/\underline{\leftrightarrow}$ can be restricted to form the smaller models.

3.3.2 Some example specifications in ACP

Now we will give two examples of specification in ACP that correspond to previous examples from CCS: (1) two one-cell buffers connected together, and (2) the alternating bit protocol. [14]

A one-cell buffer, with input port labelled "1" and output port labelled "2", buffering elements of some finite data set D, may be specified as:

$$B^{12} \quad = \quad \sum_{d \in D} r_1(d) \cdot s_2(d) \cdot B^{12}$$

(The names of the ports of the process are in superscript.) Likewise, a one-cell buffer with input port "2" and output port "3" would be specified as:

$$B^{23} \quad = \quad \sum_{d \in D} r_2(d) \cdot s_3(d) \cdot B^{23}$$

A buffer of capacity 2, with input port "1" and output port "3", can be specified with two equations:

$$B_2^{13} \quad = \quad \sum_{d \in D} r_1(d) \cdot B_d$$

$$B_d \quad = \quad s_3(d) \cdot B_2^{13} + \sum_{e \in D} r_1(e) \cdot s_3(d) \cdot B_e .$$

The buffers B^{12} and B^{23} will be joined together and compared to the two-element buffer. Just as in CCS, is is desired to encapsulate the two "halves" of communication at the internal ports. This is done in ACP by setting $H = \{r_2(d), s_2(d) : d \in D\}$ and considering the process $\partial_H(B^{12} \parallel B^{23})$. This simple expression can now be manipulated in accordance with the axioms of ACP to derive a set of recursive equations for this process using process variables X and X_d (for $d \in D$).

$$
\begin{aligned}
X \quad &\stackrel{def}{=} \quad \partial_H(B^{12} \parallel B^{23}) \\
&= \quad \partial_H(B^{12} \parallel\!\!\!\perp B^{23}) + \partial_H(B^{23} \parallel\!\!\!\perp B^{12}) + \partial_H(B^{12} \mid B^{23}) \\
&= \quad \partial_H(\sum_{d \in D} r_1(d) \cdot (s_2(d)B^{12} \parallel B^{23})) + \delta + \delta \\
&= \quad \sum_{d \in D} r_1(d) \cdot \partial_H(s_2(d)B^{12} \parallel\!\!\!\perp B^{23} + B^{23} \parallel\!\!\!\perp s_2(d)B^{12} + s_2(d)B^{12} \mid B^{23}) \\
&= \quad \sum_{d \in D} r1(d) \cdot (\delta + \delta + c_2(d) \cdot \partial_H(B^{12} \parallel s_3(d)B^{23})) \\
&= \quad \sum_{d \in D} r1(d) \cdot c_2(d) \cdot X_d
\end{aligned}
$$

---

[14]Unlike before, we now include the data to be transmitted in the specification.
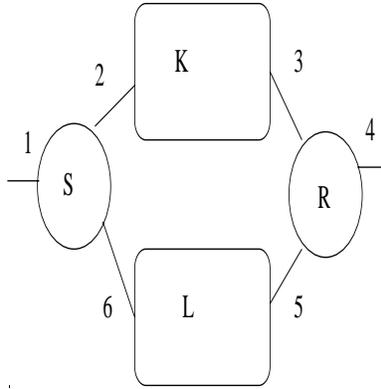
Figure 6: Alternating bit protocol for the ACP specification

and

$$
\begin{aligned}
X_d &\stackrel{def}{=} \partial_H(B^{12} \parallel s_3(d)B^{23}) \\
&= \partial_H(B^{12} \underline{\parallel} s_3(d)B^{23}) + \partial_H(s_3(d)B^{23} \underline{\parallel} B^{12}) + \partial_H(B^{12} \mid s_3(d)B^{23}) \\
&= \sum_{e \in D} r_1(e) \cdot \partial_H(s_2(e)B^{12} \parallel s_3(d)B^{23} + s_3(d) \cdot \partial_H(B^{12} \parallel B^{23}) + \delta \\
&= \sum_{e \in D} r_1(e) \cdot (\delta + s_3(d) \cdot \partial_H(s_2(e)B^{12} \parallel B^{23}) + \delta) + s_3(d) \cdot X \\
&= \sum_{e \in D} r_1(e) \cdot s_3(d) \cdot (\delta + \delta + c_2(e) \cdot \partial_H(B^{12} \parallel s_3(e)B^{23})) + s_3(d) \cdot X \\
&= \sum_{e \in D} r_1(e) \cdot s_3(d) \cdot c_2(e) \cdot X_e + s_3(d) \cdot X
\end{aligned}
$$

Thus the result of all this manipulation is that $\partial_H(B^{12} \parallel B^{23})$ is equivalent to the recursive specification:

$$
\begin{aligned}
X &= \sum_{d \in D} r_1(d) \cdot c_2(d) \cdot X_d \\
X_d &= s_3(d) \cdot X + \sum_{e \in D} r_1(e) \cdot s_3(d) \cdot c_2(e) \cdot X_e
\end{aligned}
$$

A comparison of this specification with the previous one for the two-element buffer (B_2^{13}) shows that they are identical except for the internal actions $c_2(d)$. This shows the effect of the separation out of encapsulation from abstraction. Although the actions on the internal port have been, in a sense "isolated", they are not invisible to external processes.

The alternating bit protocol specification will use the port labelling as shown in Figure 6. The goal is to define processes S, K, L, R such that the behavior of the entire process, aside from the communications at the internal ports 2,3,5,6, behaves as a one-element buffer and so satisfies the equation:

$$
B^{14} = \sum_{d \in D} r_1(d) \cdot s_4(d) \cdot B^{14}
$$

Let D be the finite data set and define the set of frames of data by $F = \{d0, d1 : d \in D\}$. The

channels K and L are defined as follows: [15]

$$K \;=\; \sum_{x \in F} r_2(x)(i \cdot s_3(x) + i \cdot s_3(\perp)) \cdot K$$

$$L \;=\; \sum_{n=0,1} r_5(n)(i \cdot s_6(n) + i \cdot s_6(\perp)) \cdot L$$

The atom i is used to make the choice non-deterministic so that the decision whether or not the frame will be corrupted is internal to K or L. Note that unlike the example in CCS, the data is not duplicated or lost, merely corrupted. The sender S is defined as follows ($n = 0, 1, d \in D$):

$$S \;=\; S0 \cdot S1 \cdot S$$

$$Sn \;=\; \sum_{d \in D} r_1(d) \cdot Sn_d$$

$$Sn_d \;=\; s_2(dn) \cdot Tn_d$$

$$Tn_d \;=\; (r_6(1-n) + r_6(\perp)) \cdot Sn_d + r_6(n)$$
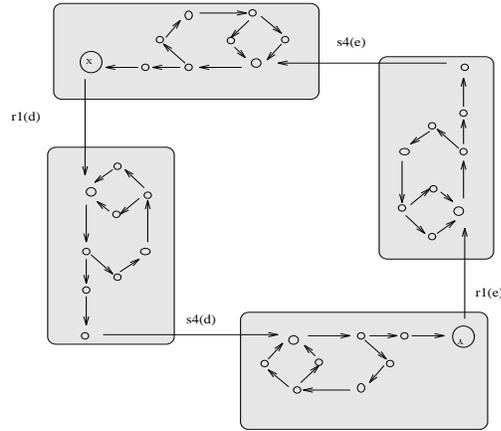
The receiver R is defined as follows ($n = 0, 1$):

$$R \;=\; R1 \cdot R0 \cdot R$$

$$R_n \;=\; (\sum_{d \in D} r_3(dn) + r_3(\perp)) \cdot s_5(n) \cdot R_n + \sum_{d \in D} r_3(d(1-n)) \cdot s_4(d) \cdot s_5(1-n)$$

Note the slight difference in the definition of the ABP in this case as contrasted with the CCS example. Since no timer is being used, retransmission is triggered by receiving a bad acknowledgement. The composition of these four processes is $\partial_H(S \parallel K \parallel L \parallel R)$, where $H = \{r_k(x), s_k(x) : x \in F \cup \{0, 1, \perp\}, k = 2, 3, 5, 6\}$ (the internal actions). Recursive equations can be derived for this process using the ACP axioms. The calculations are quite tedious and long (basically using the expansion law and axioms of encapsulation) and are omitted here. The equations are defined in terms of the following abbreviations (for every $d \in D$):

$$X \;=\; \partial_H(S \parallel K \parallel L \parallel R)$$

$$X1_d \;=\; \partial_H(S0_d \cdot S \parallel K \parallel L \parallel R)$$

$$X2_d \;=\; \partial_H(T0_d \cdot S \parallel K \parallel L \parallel s_5(0) \cdot R0 \cdot R)$$

$$Y \;=\; \partial_H(S1 \cdot S \parallel K \parallel L \parallel R0 \cdot R)$$

$$Y1_d \;=\; \partial_H(S1_d \cdot S \parallel K \parallel L \parallel R0 \cdot R)$$

$$Y2_d \;=\; \partial_H(T1_d \cdot S \parallel K \parallel L \parallel s_5(1) \cdot R)$$

The resulting recursive specification is shown in Figure 7 In order to make sense of what these recursive variables refer to, see Figure 8, which shows a process graph for one data element for X and Y. Note that the symmetry of the graph simply reflects processes that are the same except for the current bit being used for everification. Before going on with a verification of this protocol, it's worth noting that it is clearly a mess. With the previous example of the linked buffers, it was easy to see that it was equivalent to a two-element buffer if one could "mentally abstract" away from the internal port. That's obviously more difficult in this case. What is desired is to formalize abstraction such that Figure 8 looks more like Figure 9, where only the action on the external ports 1 and 4 are seen, and the four individual segments that cycle only within themselves are hidden.

---

[15]$\perp$ represents an error message; it is assumed that an incorrect transmission can be recognized.

$$\partial_H(S \parallel K \parallel L \parallel R) \ =$$

$$X \ = \ \sum_{d \in D} r_1(d) \cdot X1(d)$$

$$X1_d \ = \ c_2(d0)(i \cdot c_3(\bot) \cdot c_5(1)(i \cdot c_6(\bot) + i \cdot c_6(1)) \cdot X1_d + i \cdot c_3(d0) \cdot s_4(d) \cdot X2_d)$$

$$X2_d \ = \ c_5(0)(i \cdot c_6(\bot) \cdot c_2(d0)(i \cdot c_3(\bot) + i \cdot c_3(d0))X2_d + i \cdot c_6(0) \cdot Y)$$

$$Y \ = \ \sum_{d \in D} r_1(d) \cdot Y1(d)$$

$$Y1_d \ = \ c_2(d1)(i \cdot c_3(\bot) \cdot c_5(0)(i \cdot c_6(\bot) + i \cdot c_6(0)) \cdot Y1_d + i \cdot c_3(d1) \cdot s_4(d) \cdot Y2_d)$$

$$Y2_d \ = \ c_5(1)(i \cdot c_6(\bot) \cdot c_2(d1)(i \cdot c_3(\bot) + i \cdot c_3(d1))Y2_d + i \cdot c_6(1) \cdot X)$$

Figure 7: Specification for $\partial_H(S \parallel K \parallel L \parallel R)$

Figure 8: Graph for alternating bit protocol in ACP

Figure 9: Desired abstraction for alternating bit protocol in ACP

3.4 ACP with abstraction (ACP^τ)

The separation of encapsulation and abstraction, while perhaps not as intuitive and simple as in CCS, appears to allow a greater refinement of abstraction possibilities. Consider again Figure 9. If the set of internal steps is $I = \{c_k(x) : x \in F \cup \{0, 1, \perp\}, k = 2, 3, 5, 6\} \cup \{i\}$, then only the actions $r_1(d)$ and $s_4(d)$ are external. What is desired is to have an *abstraction operator* τ_I, making internal steps invisible, such that $\tau_I(X) = B^{14}$.

Thus, the silent step τ and abstraction operator τ_I are added to ACP, along with the following new axioms:

$$
\begin{array}{ll}
x\tau = x & B1 \\
x(\tau(y + z) + y) = x(y + z) & B2 \\
\tau_I(a) = a \text{ if a} \notin I & TI1 \\
\tau_I(a) = \tau \text{ if a} \in I & TI2 \\
\tau_I(x + y) = \tau_I(x) + \tau_I(y) & TI3 \\
\tau_I(xy) = \tau_I(x) \cdot \tau_I(y) & TI4
\end{array}
$$

3.4.1   models for ACP^τ

Just as with CCS, the introduction of τ-actions leads to the possibilities for various definitions of bisimilarities on process graphs. What Baeten & Weijland call *branching simulation*, written $\underleftrightarrow{}_b$, can be thought of as corresponding to CCS's weak bisimulation. Also, just as weak bisimulation was not a congruence relation with respect to CCS, branching bisimulation is not a congruence relation with respect to ACP. Thus, *rooted branching (rb) bisimulation*, written $\underleftrightarrow{}_{rb}$, is introduced in ACP^τ just as = was introduced in CCS. The graph model $\mathbf{G}_\infty$ is easily altered to handle abstraction by stipulating that for any graph g, the graph $\tau_I(g)$ is created by replacing all labels from I by τ. It can be shown that $\mathbf{G}/\underleftrightarrow{}_{rb} \models ACP^\tau$.

3.4.2   Example verification in ACP^τ

In order to complete the verification of the alternating bit protocol, we need to show that

$$\tau_I(I) \circ \partial_H(S \parallel K \parallel L \parallel R) = B^{14}.$$

From the definition of B^{14}, this is equivalent to showing that:

$$\tau_I(I) \circ \partial_H(S \parallel K \parallel L \parallel R) \;=\; (\sum_{d \in D} r_1(d) \cdot s_4(d) \cdot \tau_I \circ \partial_H(S \parallel K \parallel L \parallel R)).$$

In order to accomplish this, a rule called the *Cluster Fair Abstraction Rule ($CFAR^b$)* is developed based on the ACP^τ axioms, assuming branching bisimulation, that allows the grouping together of internal cycles such as in the ACP graph. For example, the *cluster* around $X1_d$ is defined as follows (refer to Figure 8:

$$\begin{aligned}
X1_d &= c_2(d0) \cdot Z_1 \\
Z_1 &= i \cdot Z_2 + i \cdot c_3(d0) \cdot s_4(d) \cdot X2_d \\
Z_2 &= c3(\perp) \cdot Z_3 \\
Z_3 &= c5(1) \cdot Z_4 \\
Z_4 &= i \cdot Z_5 + i \cdot Z_6 \\
Z_5 &= c6(1) \cdot X1_d \\
Z_6 &= c6(\perp) \cdot X1_d
\end{aligned}$$

Then $\{X1_d, Z_1, Z_2, Z_3, Z_4, Z_5, Z_5, Z_5\}$ is a cluster and from $CFAR^b$ it can be derived that:

$$\begin{aligned}
\tau_I(X1_d) &= \tau \cdot \tau_I(i \cdot c3(d0) \cdot s_4(d) \cdot X2_d) \\
&= \tau \cdot s_4(d) \cdot \tau_I(X2_d)
\end{aligned}$$

In other words, this means that $X1_d$ may cycle within itself some number of times before finally sending data on the s_4 line and entering state $X2$. Similarly, the cluster around $X2_d$ is reduced to get:

$$\begin{aligned}
\tau_I(X2_d) &= \tau \cdot \tau_I(i \cdot c_6(0) \cdot Y) \\
&= \tau \cdot \tau_I(Y).
\end{aligned}$$

After some more equational manipulation, which is omitted here, the results are:

$$\begin{aligned}
\tau_I(X) &= \sum_{d \in D} r_1(d) \cdot s_4(d) \cdot \tau_I(Y) \\
\tau_I(Y) &= \sum_{d \in D} r_1(d) \cdot s_4(d) \cdot \tau_I(X)
\end{aligned}$$

It can be shown that it follows by properties of recursive equations in ACP^τ that $\tau_I(X) = \tau_I(Y)$ and so the definition of B^{14} is satisfied by $\tau_I(X)$.

The definition of the rule $CFAR^b$ is specifically defined on the assumption that the choices made by the channels are fair - that is, no channel is completely defective and corrupts a message infinitely many times in a row. This was also the assumption in the CCS specification of ABP. The difference is that Baeten & Weijland also describe, in great detail, a different rule and its consequences should

such a fairness condition does not hold. The greater degree of formal rigor in ACP^τ is typical of the difference in the two approaches.

The difference in approaches to bisimulation equivalence is worth noting. In CCS, an extensive comparison of nodes in derivation trees is required, whereas in ACP^τ the same results are accomplished via equational manipulations. It would be interesting to compare attempts at automation of bisimulation searching for these two approaches. As mentioned, CCS might be more easily automated in a "brute force" fashion, while ACP^τ might require more sophisticated proof techniques. Although not discussed in this paper, there is also an equational theory for CCS, but much more emphasis is put on such a system in ACP^τ.

4   Decidability

Since bisimulation is a crucial issue in process verification, the question of decidability of whether two processes are bisimilar is obviously of interest. There is another point of view, however, from which to view this question. In addition to its use for process specification, process calculi can be thought of as a successor to automata theory, with the main difference of course being to look at the behavior of the processes, and not just their execution traces. From that perspective, a reconsideration of automata theory results in this new framework is of interest, and it is from this perspective that the proof in the next section proceeds.

4.1   Decidability of Processes Generating Context-Free Languages

It is a well-known result in automata theory that the question of equivalence between context-free languages is undecidable. In remarkable contrast to this result, it has been shown [Baeten et al. 1993] that when CFLs are examined in a process calculi framework, the bisimulation equivalence of those processes is decidable.

4.1.1   Encoding of CFL's in BPA

The process calculus that will be used for the encoding of context-free languages is BPA (without δ), as described in section (3.1). Before describing a translation of a CFG G into a member of the model $\mathbf{G}/\underline{\leftrightarrow}$ consisting of finitely branching process graphs (see section (3.1.1)), two concepts are needed first:

finite trace set - Each process graph g has a *finite trace set*, written $ftr(g)$. An element of $ftr(g)$ consists of all the actions from the root to a termination node.

norm The *norm* of a process graph g, written $|g|$, is the least number of steps it takes from the root to reach a termination node, if any such node is reachable. That is, $|g|$ is the minimum length of a completed finite trace of g. The norm of a node s in a process graph g, written $|s|$, is the norm of the subgraph determined by s. The norm of a process p is the norm of the representing process graph, and a process is *normed* if every subprocess has a norm. [16] What this essentially means is that there are no superfluous parts of the graph that do not contribute to the generation of finite traces.

---

[16]Process q is called a subprocess of process p if p, q have representing process graphs g, h, respectively, such that h is a subgraph of g.
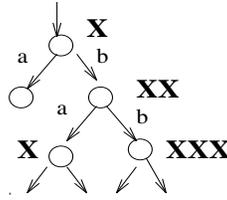
Figure 10: The process graph for $X = a + bXX$

Proposition 13 *Every CFL (without ε) is the finite trace set of a normed process p, recursively defined by means of a guarded system of recursion equations in restricted GNF.*

Sketch of proof. Consider some CFL L generated by a grammar CFG G. Convert G into G', where the latter is in restricted GNF form. [17] A trivial notational change to G' results in the recursive specification $E^{G'}$ in the language of BPA. [18] Although $E^{G'}$ now defines a process in BPA, that process may not be normed. Baeten et al. give a detailed description of how to translate $E^{G'}$ into $E'^{G'}$, where the latter is normed. For example, the system (1) gets converted to system (2):

$$E = \{X = aY + bXZ + cXX, \quad Y = d + cYY, \quad Z = aZ + bYZ\} \tag{1}$$

$$E = \{X = aY + cXX, \quad Y = d + eYY \quad \} \tag{2}$$

This is, of course, just the usual procedure for eliminating useless variables and productions from a CFG (see, for example [Hopcraft & Ullman 1979]), placed in the context of BPA recursion equations. Since E'^G is a guarded system, it will have one solution, namely the process graph p. The finite trace set of p will be exactly the CFL generated by G'. This is because every path from the root of p to a terminating node is a leftmost derivation in G'. For a simple example, consider the graph for the one-variable recursive specification $E = \{X = a + bXX\}$ in Figure 10, where each node is labelled with the process that remains to be done at that node. For example, $bbaaa \in ftr(p)$ because $X \Rightarrow bXX \Rightarrow bbXXX \Rightarrow bbaXX \Rightarrow bbaaX \Rightarrow bbaaa$. Thus, proposition (13) states that the set of irredundant CFG's corresponds exactly with the set of normed processes in BPA. The goal of the rest of the proof is to prove that the the bisimulation equivalence of two normed systems of recursion equations is decidable. This essential idea behind the proof is that the process graph of any normed process exhibits a certain periodic regularity. The same structural patterns in the graph get repeated throughout, and crucially there are only a finite number of such patterns. Thus, for any two such graphs, there will be a certain "level" k ("level" will be precisely defined), at which all the structural patterns that will ever appear in the graphs have already appeared. It is shown that if there is no bisimulation up to level k, then there is no bisimulation at all. Since k is computable, and for any k there are only a finite number of possible bisimulations (since only a finite number of nodes are being compared), the decidability follows.

---

[17] A CFG in which every production is of the form $A \to a\alpha$, where A is a variable, a is a terminal, α is a possibly empty string of variables, is said to be in *Greibach Normal Form GNF*. If moreover the length of α does not exceed 2, then the CFG is in *restricted* GNF form. It's a known theorem that every CFG that does not generate ε can be rewritten as a CFG in restricted GNF that is weakly equivalent to the original grammar,

[18] By replacing composition + with | and $\to$ with =
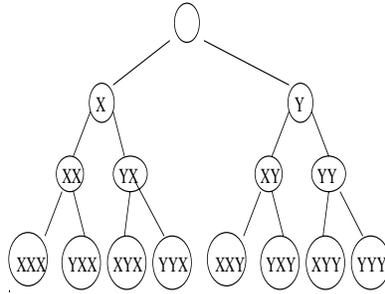
Figure 11: $t(E)$ for $X^* = \{X, Y\}$

4.1.2   Universal Tree & Translation Equivalence

Much of the detail of the proof is devoted to explicitly capturing the periodicity of the graph. Two of the most important concepts for this purpose are:

Universal Tree - The *universal tree* $t(E)$ is the tree having as nodes all the words $w \in X^* = \{X_1, ..., X_n\}^*$, where $X_1, ..., X_n$ are the variables used by E. The top node is the empty word (called the *termination node*), and has as children $X_1, .., X_n$. Each succeeding level is defined inductively: if w is a node of $t(E)$ then its children are $X_1 w ... X_n w$. Figure 11 shows the tree $t(E)$ for $X^* = \{X, Y\}$.

The idea of $t(E)$ is that it will serve as the underlying "node space" for the process graph $g(E)$ determined by E. Any process graph can be thought of as being overlaid on top of $t(E)$, and so several concepts that follow are defined in terms of $t(E)$ rather than a particular graph. A process graph may not use up all of $t(E)$.

Translation Equivalence - Let $w \in X^*$. The *translation* T_w is the mapping from X^* to X^* defined by: $T_w(v) = vw$, the concatenation of v followed by w. The *inverse translation* T_w^{-1} is the partial mapping from X^* to itself that removes the postfix w. A *shift* is an inverse translation followed by a translation: $T_w T_v^{-1}$ and so a shift replaces a postfix v by a postfix w.

Let $V, W \subseteq X^*$ and suppose that for some U and v, w we have: $T_v(U) = V, T_w(U) = W$. Then V, W are *equivalent modulo translation*, written $V \equiv_T W$, meaning that V, W differ by a shift. $\equiv_T$ can be shown to be an equivalence relation.

As will be seen, translation equivalence is used to capture the relationship between repeated occurrences of the same structure in a graph.

Some more definitions:

length - For $w \in X^*$, the *length* of w, $lth(w)$, is the number of symbols in w.

distance - For $v, w \in X^*$, the *distance* $d(v,w)$ between v and w is the minimum number of steps(edges) necessary to go from v to w in $t(E)$, where E has variables **X**. An equivalent definition is: Let u be the maximal common postfix of v, w, and $v = v'u$ and $w = w'u$. then $d(v, w) = lth(v') + lth(w')$.

far apart - For $v, w \in X^*$, v and w are *far apart* if $d(v, w) \geq 3$. If $V, W \subseteq X^*$, then the sets V, W are far apart if all pairs $v \in V, w \in W$ are far apart.
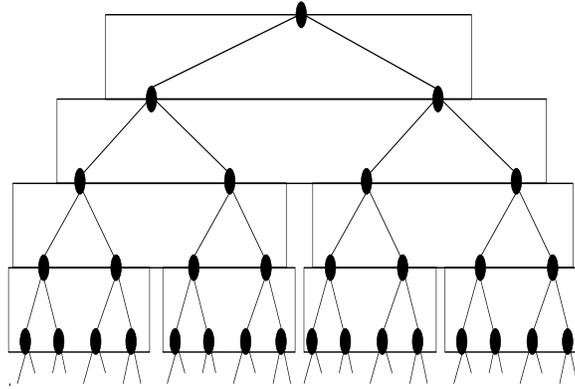
Figure 12: A slicing of $t(E)$

sphere d - The *sphere with center w and radius r, $B(w,r)$, is* the subset of X^*, in $t(E)$, consisting
of all v whose distance to w does not exceed r. Also, $\mathbf{B}_r$ is the collection all spheres with a
fixed radius r.

uniformly bounded - If $\mathbf{V} = \{V_i | i \in I\}$ is a collection of subsets of X^*, and: $\exists r \forall i \exists w V_i \subseteq B(w,r)$,
then the elements of $\mathbf{V}$ are *uniformly bounded*.

The two most important results obtained using these concepts are:

Proposition 14 *Let $\mathbf{V}$ be a uniformly bounded collection of subsets of X^*. Then $\mathbf{V}$ is finitely
partitioned by translation equivalence.*

Baeten et al. do not give a rigorous proof, but instead attempt to motivate a feeling for why this is
valid. Consider the set $\mathbf{B}_1$ of all spheres with radius one in the the example universal tree shown
in Figure 11. Now consider the set Z consisting of the three spheres $B(\varepsilon, 1)$, $B(X, 1)$, and $B(Y, 1)$.
Every sphere with radius one is translation equivalent to one of the spheres in Z. For example,
the sphere $B(YX, 1) = \{X, YX, XYX, YYX\} \equiv_T B(Y, 1) = \{\varepsilon, Y, XY, YY\}$. So the set $\mathbf{B}_1$ can be
partitioned into a finite number of equivalence classes, where each class corresponds to one of the
members of Z. The same idea should work for any uniformly bounded collection of subsets of X^*.

Proposition 15 *Let W be a subset of X^*, such that:*
(i) $\exists c_1, c_2 \in \mathcal{N} \; \forall w \in W c_1 \leq lth(w) \leq c_2$
(ii) W cannot be partitioned into W_1, W_2 which are far apart.
Then W is contained in a sphere $B(w,r)$ where r depends only on c_1, c_2.

Proof (sketch). The proof of this given by Baeten et al. is ambiguous and quite unsatisfactory,
although I think that the proposition itself is valid. A detailed examination of this proposition will
detract from an overall understanding of the proof, so I will attempt, with a figure, to illustrate the
motivation for this proposition. Consider Figure 12, which shows $t(E)$ from Figure 11, divided into
slices of thickness one. Each rectangle indicates one W that cannot be partitioned into subparts
W_1, W_2 that are far apart. According to prop (15), this collection of rectangles is uniformly
bounded (by definition of the latter), and so by prop (14) this collection is finitely partitioned by
translation equivalence. The same could be done for slices of different "thickness".

$$
\begin{aligned}
X &= a + bY + fXY \\
Y &= cX + dZ \\
Z &= gX + eXZ
\end{aligned}
$$

Figure 13: An example recursive specification

Propositions (14) and (15) are key results for the entire proof, because in section (4.1.4) an arbitrary process graph will be sliced up into *principal fragments* that are far apart and thus shown to be finitely partitioned by translation equivalence. The next part of the proof, in section (4.1.3), is dedicated to using the results of these two propositions in the context of an actual process graph instead of $t(E)$, and then in section 4.1.4 those revised versions of the propositions will be used to obtain the important *regular decomposition* result.

Note that since by assumption E is normed, then by definition of normed, each $w \in X^*$ has a norm $\mid w \mid$, in addition to its $lth(w)$. It can be shown (proof omitted) that prop (15) remains valid with $lth(w)$ replaced by $\mid w \mid$, and this is the form in which the proposition will be used later in the proof.

4.1.3   The Process Graph & Process Graph Fragment

The *process graph* $g(E)$ for the system E has so far been considered as graph of the possible transitions. However, it can also be thought of by first creating $t(E)$ for the variables X in E, filling in labelled edges in $t(E)$, and then deleting parts of the graph that are inaccessible from the root node. Note that although a process graph is not a tree, it nevertheless exhibits, from a more global point-of-view, a certain "tree-like" structure. For example, look at Figure 14, which is a partial process graph for the system shown in Figure 14. Note also that in Figure 14, the the norms are "respected graphically" - that is, a node with norm n will be positioned on level n.

The notion of *process graph fragment* is aimed at capturing these repeating patterns: Let E be a system of recursion equations with variables $\mathbf{X} = \{X_1, ..., X_n\}$ and action alphabet $A(E)$.

Process Graph Fragment - A *(process) graph fragment* in the space $t(E)$ consists of some subset N of nodes of X^* together with some edges $w \rightarrow_a v (w, v \in N)$ labelled by atoms in $A(E)$. $\alpha, \beta, ..$ will be used to denote graph fragments.

Two notions previously used are updated for use in the context of graph fragments:

weakly fragmented A graph fragment is *weakly connected* if it cannot be partitioned into two graph fragments which are far apart.

translations *Translations* T_w of graph fragments are defined as for node sets, with the extra requirement that a translation also respects labelled edges.

Proposition 16 *Let α be a graph fragment of $g(E)$ such that*
(i) $\exists c_1, c_2 \in \mathcal{N} \ \forall w \in \alpha \ c_1 \leq \mid w \mid \ \leq c_2$, and
(ii) α is weakly connected.
Then α is contained in a sphere $B(w, r)$ where r depends only on c_1, c_2, and E (in a computable way).
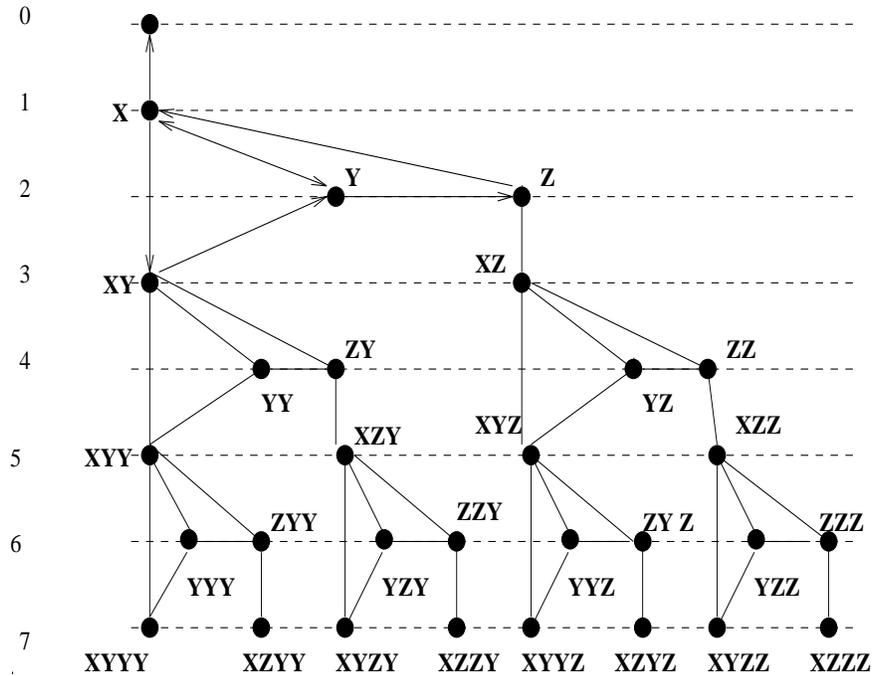
Figure 14: A partial process graph.

Proof: From proposition (15).

Proposition 17 *Let $(\alpha_i)_{i \in I}$ be a collection of fragments of $g(E)$, and let the α_i be uniformly bounded. Then the collection is finitely partitioned by translation equivalence. Moreover, the number of elements of the partition can be computed from E.*

Proof: Since the collection is uniformly bounded, it follows from prop (14) the collection is finitely partitioned by translation equivalence. The "computable" part of the proof is very vague, since it depends on the proof of prop(14), which itself is very vague. It most likely is supposed to refer to a calculation of the number of equivalence sets.

4.1.4 The Regular Decomposition of the Process Graph

In this section a decomposition of a process graph into *slices* and *principal fragments* will be defined. Propositions (16) and (17) allow us to show that for this decomposition there are only a finite number of such fragments modulo translation equivalence. This not quite adequate, however, because it also needs to be shown that these fragments are not in some haphazard layout in the graph, but instead make up a *regular* tree-like structure. The concept of a *regular decomposition* is used to capture this notion:

regular tree A node-labelled tree is *regular* if there are (modulo isomorphism of node-labelled trees) only finitely many subtrees. The labels in this case will be very complicated - translation equivalence classes of process graph fragments.

Regular Decomposition A *regular decomposition* of the process graph $g(E)$ is a tree $\mathcal{T}$ where each node s is labelled with a graph fragment α_s, such that

1. each α_s is a finite graph fragment in $t(E)$.

2. the union of all α_s is $g(E)$.

3. for nodes s, t in $\mathcal{T}$, α_s and α_t are disjoint iff s, t are not connected by a single edge in $\mathcal{T}$.

4. the collection of all α_s (all nodes s in $\mathcal{T}$) is finitely partitioned by translation equivalence.

5. if $\underline{\alpha_1}, ..., \underline{\alpha_k}$ denote the finitely many equivalence classes in which the α_s are partitioned, and each label α_s is replaced by the label denoting its equivalence class, the resulting node-labelled tree $\mathcal{T}'$ is regular.

A decomposition for any process graph $g(E)$ is defined as follows, and this decomposition will be shown to be regular:

1. $g(E)$ will be divided into fragments, called *slices*, numbered 0,1,2,... Each slice has thickness d, and d is called the *amplitude* of the decomposition.

2. The nth slice contains the nodes s of $g(E)$ with $nd \leq |s| \leq (n+1)d$ and also those nodes reachable by one step in $g(E)$ from a node s with $nd \leq |s|(n+1)d$. [19]

3. The nth slice is now the fragment of $g(E)$ obtained by taking the restriction of $g(E)$ to the set of nodes of the nth slice.

4. The nodes of the nth slice will be partitioned into equivalence classes as follows: define for nodes s, t in the nth slice: $s \leftrightarrow t$ iff s, t have distance 0,1,2, or 3. Let $\Leftrightarrow$ be the transitive closure of $\leftrightarrow$. This is an equivalence relation on the nodes of the nth slice, partitioning these nodes into equivalence classes denoted by $[s]_{\Leftrightarrow}$.

5. The restriction of $g(E)$ to the set of nodes $[s]_{\Leftrightarrow}$ in slice n, is called a *principal fragment*.

Proposition 18 *Let $g(E)$ be divided in slices. Then the corresponding principal fragments of $g(E)$ are uniformly bounded, and thus finitely partitioned by translation equivalence. Moreover, the number of principal fragments of $g(E)$ can be computed from E.*

Proof:

1. By the definition of a principal fragment, all principal fragments of a slice n are far apart.

2. By prop (15), the collection of all principal fragments (of all slices) of $g(E)$ is a uniformly bounded collection.

3. By prop (17), the collection of principal fragments is finitely partitioned by translation equivalence, and the number of elements is computable from E.

The following proposition is needed to prove theorem (20):

Proposition 19 *Let α and α' be fragments of $g(E)$, which are translation equivalent. Let s be a node in α that is not minimal in α. Suppose $s \rightarrow_a t$ is an edge such that $\alpha \cup \{s \rightarrow_a t\}$ is again a fragment of $g(E)$. Let s' be the point in α' corresponding (after the same shift as from α to α') to s.*

Then there is a t' and an edge $s' \rightarrow_a t'$ such that $\alpha' \cup \{s' \rightarrow_a t'\}$ is also a fragment of $g(E)$; moreover, the two extended fragments are again translation equivalent by the same shift.

---

[19]There is no explanation for this extra clause ("and also..."). I am not sure why it is needed
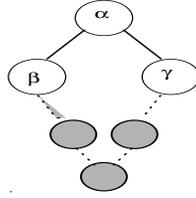
Figure 15: No possible confluence in the decomposition

Proof: details omitted. It's a straightforward proof based on the properties of translation equivalent fragments.

Theorem 20 *Let E be a normed system of recursion equations in restricted GNF, in the signature of BPA, and let $g(E)$ be the corresponding normed process graph. Then $g(E)$ has a regular decomposition; moreover, the amplitude d of the decomposition can be chosen arbitrarily such that $d \geq c(E)$ for some constant $c(E)$ computable from E.*

Proof. This theorem is the culmination of the proof so far, and follows in a mostly direct manner from the work so far:

1. A tree of graph fragments can be created, and it is guaranteed to be a well-formed tree because no "confluence" can occur, as in Figure 15. This is because by the very definition of a graph fragment, all the points of β and γ are far apart, and so going downwards from such points only increases the distance, and so no confluence of lower principal fragments is possible.

2. From propositions (16) and (17), it follows that there are only finitely many labels (fragments) modulo translation equivalence.

3. All that remains is to show the regularity of the decomposition. Consider two nodes s, t in $\mathcal{T}$ occupied by α_s, α_t, with $\alpha_s \equiv_T \alpha_t$. Let $\mathcal{T}_s$, $\mathcal{T}_t$ be the subtrees of $\mathcal{T}$ determined by s, t respectively. Let G_s, G_t be the graph fragments of $g(E)$ obtained by taking the unions of all the labels in $\mathcal{T}_s$ and $\mathcal{T}_t$, respectively. Then it needs to be shown that $G_s \equiv_T G_t$. This follows from repeated application of prop (19).

The one questionable part of this theorem is in the statement that the "amplitude d of the decomposition can be chosen arbitrarily such that $d \geq c(E)$ for some constant $c(E)$ computable from E." It is unclear as to what this restriction on d would be.

Remark: It is surprising to note that so far the restriction to normed process graphs has not been crucial. All the proofs so far will carry over if the length lth is used instead of the norm. So in fact the following more general version of theorem(20) holds:

Theorem 21 *Let E be a system of recursion equations in BPA in restricted GNF. Then the corresponding graph $g(E)$ has a regular decomposition.*

4.1.5   The main result

Some definitions: Let E_1, E_2 be normed systems of recursion equations in restricted GNF.

n-prefix If R is a bisimulation between $g(E_1)$, $g(E_2)$, then the *prefix up to n*, or *n-prefix* is the restriction of R to the nodes of g, h whose level does not exceed n.

partial bisimulation up to level n - A *partial bisimulation* between $g(E_1)$, $g(E_2)$ *up to level n* is a relation R whose domain consists of the nodes of $g(E_1)$ with level $\leq n$, and whose codomain consists of the nodes of $g(E_2)$ with level $\leq n$, such that R is a bisimulation.

d-sufficient - Suppose that $g(E_1)$ and $g(E_2)$ have regular decompositions with common amplitude d, and let R be a partial bisimulation up to slice k. R is *d-sufficient* if the following holds true:

> Suppose that α is a fragment of slice k in $g(E_1)$, and β is a fragment of slice k in $g(E_2)$. Also, the successor fragments of α are $\alpha_1, ..., \alpha_n$ and the successor fragments of β are $\beta_1, ...\beta_m$ for some n, m. Suppose also that fragments α, β are related by the partial simulation R and that at least one slice higher (that is, a slice with level $< k$), there are translation equivalent copies α', β' of α, β (which then must have children $\alpha_1', ..., \alpha_n'$ and $\beta_1', ..., \beta_m'$) such that the restriction of R to $\alpha \times \beta$ coincides, modulo translation equivalence $\equiv_T$, with the restriction of R to $\alpha' \times \beta'$. If for each pair α, β in the kth slice such a copy α', β' exists, then the partial simulation R is called *d-sufficient*.

> In other words, d-sufficiency is a formalization of the idea that if there are only a finite number of patterns that need to be related under a bisimulation, then at a certain level all such patterns will have appeared. It is easy to show that if a partial simulation R is d-sufficient, then it can be extended to a total bisimulation.

Theorem 22 *Let $g(E_1)$, $g(E_2)$ be process graphs, each with regular decompositions of common amplitude d, and let R be a bisimulation between them. Then R has a d-sufficient M-prefix for each $M \geq N(E_1, E_2, d)$, where $N(E_1, E_2, d)$ is some constant computable from E_1, E_2, and d.*

Proof(sketch): The proof given by Baeten et al. is again very vague, but the idea appears to be this: since $g(E_1)$ and $g(E_2)$ both have regular decompositions, then there are only a finite number of graph fragments modulo translation equivalence and so there are only finitely possible relations $(\alpha \times \beta) \cap R$. Thus, there must be a certain level N (computable in some vague way from E_1, E_2, d) such that all such relations have already appeared, and so any level $M \geq N$ must be d-sufficient.

Theorem 23 *(i) Let E_1, E_2 be normed systems of recursion equations (over BPA) in restricted GNF. Then the bisimilarity relation $g(E_1) \underline{\leftrightarrow} g(E_2)$ is decidable.*
(ii) Equality of recursively defined normed processes in the graph model G of BPA is decidable.

Proof: (i) Let $g(E_1)$, $g(E_2)$ be the process graphs for E_1, E_2. Then according to theorem (20), they each have a regular decomposition, with a common amplitude d (where $d \geq c(E_1)$ and $d \geq c(E_2)$, for some constants $c(E_1)$ and $c(E_2)$ computed from E_1 and E_2, respectively). According to theorem (22), there is some computable level N such that if any bisimulation exists between $g(E_1)$ and $g(E_2)$ then there would be a d-sufficient partial bisimulation up to level N. The search space of all such partial bisimulations up to N is the set of all (finitely many) relations between the nodes of $g(E_1)$ and $g(E_2)$ up to level N. There is a bisimulation between $g(E_1)$ and $g(E_2)$ iff such a partial bisimulation is found.
(ii) This is just a rephrasing of (i).

4.1.6   Remarks

Theorem (23) is explicitly stated to be true only for *normed* systems of recursion equations.
However, up until section (4.1.5), the normed condition is irrelevant. An unclear aspect of this
proof is where exactly the normed condition is essential. Although Baeten et al. are not at all
explicit about this, it most likely has to do with the fact that if two process graphs are drawn
with their norms respected graphically (e.g., as in Figure 14), then all related pairs of nodes in a
bisimulation are horizontal connections between the two graphs.

Note that if g, h are bisimilar graphs, then $ftr(g) = ftr(h)$. but the converse is always true. In one
special case, however, that of normed, *deterministic* [20] graphs, then the converse is in fact true.
Also, a *simple* [21] CFG corresponds to a normed, deterministic graph. Since the bisimilarity of two
such CFGs is decidable, the equivalence of their finite trace sets is also decidable. Thus a corollary
of theorem (23) is another proof of the known theorem that "The equivalence problem for simple
CFLs is decidable."

All the examples grammars by Baeten et al. have a common feature: none have productions of the
form $X = aY + bY$. Grammars with such a production could probably be handled within the proof
simply by stipulating that an edge within $t(E)$ is kept if at least one production uses that edge.
Still, mention should at least have been made of this possibility.

Aside from the decidability result, the representation of the CFGs in BPA is notable. Unfortu-
nately, the usage of restricted-GNF form, while resulting in the desired property of guardedness in
the resulting BPA process definition, also severely changes the structural relation to the original
grammar, although it of course preserves the language itself. Thus, from the point-of-view of ex-
amining the structural nature of two grammars that are not in GNF form, the proof is of limited
interest. Still, it would be an interesting challenge to attempt to represent other formal language
representations in a process algebraic framework. Also, whereas the encoding in this proof repre-
sents CFGs in BPA from a *language generation* persective, encoding a *language acceptor* would be
worthwhile. Baeten et al. have a short, quite mysterious mention of this possibility:

> One can associate to push-down automata (PDAs) in a similar manner to a process;
> however as pointed out in ..., there is a PDA, even without ε and deterministic, whose
> associated graph does not display the periodicity exploited in this paper.

This proof by Baeten et al., although long and complex, was the first to use the new techniques of
process algebra to reexamine CFGs. Two obvious desired extensions to theorem (23) would be to
remove the condition on normed processes, and to include a bigger subset of process algebra. The
former was accomplished in [Christensen et al. 1992], using a completely different proof technique.
Also, a completely different, much simpler, version of the proof of theorem (23) was given in
[Hüttel & Stirling 1991]. Although it didn't extend the result of this paper, it had the advantage of
being extendable to include some aspects of concurrency. This is the subject of the section (4.2).

4.2   Decidability of Basic Parallel Processes

The decidability results in section (4.1) are concerned with a subset of ACP, one that does not use
parallel processing or communication between parallel processes. These two concepts are of course
of great importance in process calculi. Recently some positive decidability results have been found

---

[20]A process graph g is deterministic if there is no node $s \in g$ having two outgoing edges with the same label.
[21]A CFG in GNF form is simple if there is no pair of different productions $A \to a\alpha$, $A \to a\beta$.
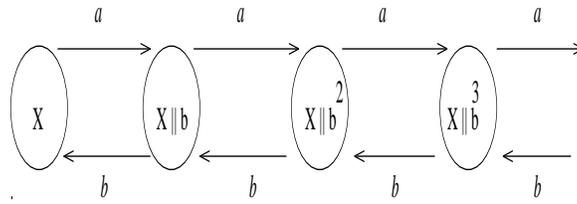
Figure 16: A sample transition graph

for processes that are defined using a parallel combinator within recursive equations. The following is based on the proof given in [Christensen et al. 1993].

4.2.1   Basic Parallel Processes

For the purposes of this proof, the authors define a class of *Basic Parallel Processes* (BPP) expressions. It consists of a countably infinite set of atomic actions $\Lambda = \{a, b, c, ...\}$ and a countably infinite set of process variables $Var = \{X, Y, Z, ...\}$, and a class of recursive equations defined by:

$$
\begin{array}{llll}
E & := & 0 & \text{(inaction)} \\
& | & X & \text{(process variable, } X \in Var) \\
& | & aE & \text{(action prefix, } a \in \Lambda) \\
& | & E + E & \text{(choice)} \\
& | & E \parallel E & \text{(merge)}
\end{array}
$$

The expression E^n represents the term $E \parallel ... \parallel E$ consisting of n copies of E combined in parallel. Also, 0-absorption allows trailing 0s to be omitted from expressions, and so the term $a0$ can be written as just a. A *BPP process* is defined by a finite family of recursive process equations

$$\Delta = \{X_i = E_i | 1 \le i \le n\}$$

where the X_i are distinct and the E_i are BPP equations containing at most the variables $Var(\Delta) = \{X_1, ..., X_n\}$. It is also assumed that each variable occurrence in the E_is are guarded, and the variable X_1 is singled out as the *leading variable* and $X_1 = E_1$ is the *leading equation*. For example, if Δ is the family $\{X = a(X \parallel b)\}$, then X generates the infinite-state transition graph in Figure 16.

Bisimilarity is defined as strong bisimilarity (since silent actions are not an issue here), and is written $\sim$. The set of finite multisets over $Var(\Delta) = \{X_1, ..., X_n\}$ is denoted by $Var(\Delta)^\otimes$ and $\alpha, \beta, ...$ are members of $Var(\Delta)^\otimes$. So each such α denotes a BPP process formed by combining the elements of α in parallel. The empty product is 0, and ordering of variables in products is ignored, so that processes denoted by elements of $Var(\Delta)^\otimes$ are identified up to associativity and commutativity of merge.

Definition 24 *A finite family* $\Delta = \{X_i = E_i | 1 \le i \le n\}$ *of guarded BPP equations is defined to be in* standard form *iff every expression E_i is of the form*

$$a_1\alpha_1 + ... + a_m\alpha_m$$

where for each j we have $\alpha_j \in Var(\Delta)^\otimes$. The empty sum is 0, and the ordering of expressions in sums is ignored, thereby defining the notion of standard form modulo associativity and commutativity of choice.

REC
$$\frac{\alpha = \beta}{\mathrm{unf}(\alpha) = \mathrm{unf}(\beta)}$$

SUM
$$\frac{\sum_{i=1}^{n} a_i \alpha_i = \sum_{j=1}^{m} b_j \beta_j}{\{a_i \alpha_i = b_{f(i)} \beta_{f(i)}\}_{i=1}^{n} \qquad \{b_j \beta_j = a_{f(i)} \alpha_{g(j)}\}_{j=1}^{m}}$$

$$where f : \{1, \ldots, n\} \to \{1, \ldots, m\}$$
$$g : \{1, \ldots, m\} \to \{1, \ldots, n\}$$

PREFIX
$$\frac{a\alpha = a\beta}{\alpha = \beta}$$

SUBL
$$\frac{\alpha \parallel \gamma = \delta}{\beta \parallel \gamma = \delta}$$
if the dominated node is labelled
$\alpha = \beta$ or $\beta = \alpha$ with $\beta \sqsubset \alpha$

SUBR
$$\frac{\delta = \alpha \parallel \gamma}{\delta = \beta \parallel \gamma}$$
if the dominated node is labelled
$\alpha = \beta$ or $\beta = \alpha$ with $\beta \sqsubset \alpha$

Figure 17: Rules of the tableau system

The authors also claim, with no proof given in the paper, that the following lemma holds:

Lemma 25 *Given any finite family of guarded BPP equations Δ we can effectively construct another finite family of BPP equations Δ' in standard form in which $\Delta \sim \Delta'$.*

For the rest of this proof, all BPP will equations under consideration will be assumed to be in standard form. The following definition is crucial to the proof:

Definition 26 *The well-founded ordering $\sqsubset$ on $Var(\Delta)^{\otimes}$ is given as follows:*

$$X_1^{k_1} \parallel \cdots \parallel X_n^{k_n} \sqsubset X_1^{l_1} \parallel \cdots \parallel X_n^{l_n}$$

iff there exists j such that $k_j < l_i$ and for all $i < j, k_i = l_i$.

Two important properties of $\sqsubset$ are:

1. it is total, meaning that for any $\alpha, \beta \in Var(\Delta)^{\otimes}$ with $\alpha \not\equiv \beta$, either $\alpha \sqsubset \beta$ or $\beta \sqsubset \alpha$.

2. $\beta \sqsubset \alpha$ implies $\beta \parallel \gamma \sqsubset \alpha \parallel \gamma$ for any $\gamma \in Var(\Delta)^{\otimes}$

4.2.2   the tableau decision method

The authors present a tableau decision method for the purpose of deciding, for any α, β of $Var(\Delta)^{\otimes}$, whether or not $\alpha \sim \beta$. The rules of the tableau system, presented in Figure 17, are built around equations $E = F$, where E and F are BPP expressions. An example tableau is shown in Figure 20 for the family of BPP processes shown in figure 19.

The basic idea of the proof is this: The rules of the tableau respect properties of bisimulation equivalence, and for some α, β, a tableau can be built up to prove whether or not $\alpha \sim \beta$. In fact,

$$\text{REC} \quad \frac{\alpha = \beta}{}$$

$$\text{SUM} \quad \frac{\sum a_i \alpha_i = \sum b_i \beta_i}{}$$

$$\text{PREFIX} \quad \frac{a_1 \alpha_1 = b_i \beta_1}{\alpha_1 = \beta_1} \quad \cdots \quad \frac{a_n \alpha_n = b_n \beta_n}{\alpha_n = \beta_n} \quad \text{PREFIX}$$

Figure 18: The schema for a basic step.

theorems (28) and (29) below show that $\alpha \sim \beta$ iff there is a successful tableau with root labelled $\alpha = \beta$. Also, the ordering $\sqsubset$ defined previously, used in conjunction with the subL and subR rules, will ensure that all tableaus are finite, and that for any α, β there are only a finite number of tableaux. The decidability follows as a consequence.

First, some terminology before an explanation of the rules: A tableau for $\alpha = \beta$ is a maximal proof tree whose root is labelled $\alpha = \beta$ and whose successive nodes are determined by application of the rules of tableau system. The rules are applied only to nodes that are not terminal. A terminal node can be either successful or unsuccessful. A successful terminal node is one labelled $\alpha = \alpha$, while an unsuccessful node is one labelled either $a\alpha = b\beta$ with $\alpha \neq \beta$ or $a\alpha = 0$ or $0 = b\beta$. A tableau is successful iff if all terminal nodes are successful. Tableaux are denoted by T (or $T(\alpha = \beta)$ to indicate the label of the root). Paths are denoted by π and nodes are denoted by **n**. If a node **n** has a label $E = F$ it may be written $\mathbf{n} : E = F$.

REC   This rule is essentially an encoding of the expansion law [22] for merge, as discussed in section (2.1.3). The notation $unf(\alpha)$ represents the *unfolding* of α, as follows:
given $Y_i = \sum_{j=1}^{n_i} a_{ij} \alpha_{ij}$ for $1 \leq i \leq m$,

$$unf(Y_1 \parallel \cdots \parallel Y_m) = \sum_{i=1}^{m} \sum_{j=1}^{n_i} \alpha_{ij}(Y_1 \parallel \cdots \parallel Y_{i-1} \parallel \alpha_{ij} \parallel Y_{i+1} \parallel \cdots \parallel Y_m).$$

SUM   After "breaking apart" an equality with REC, this rule is used to continue the bisimulation testing on the individual components. The authors do not state this, and it is not stated explicitly in the rule, but the consequents should only attempt to relate summands of α and β that begin with the same action. This is because if there was some consequent $a_i \alpha_i = b_i \beta_i$, with $a_i \neq b_i$, then it would be an unsuccessful terminal and the entire tableau would crash.

PREFIX   straightforward

Note: The REC, SUM, and PREFIX rules are used together, in components called *basic steps* from which the larger tableaux is built. The schema of a basic step for $\alpha = \beta$ is shown in Figure 18, and as can be seen consists of an application of REC to $\alpha = \beta$ followed (possibly) by an application of SUM, and followed by an application of PREFIXto each of its consequents. A basic step represents a set of single transition steps in the operational semantics: for each consequent $\alpha_i = \beta_i$ we have $\alpha \xrightarrow{a_i} \alpha_i$ and $\beta \xrightarrow{a_i} \beta_i$.

Nodes of the form $\mathbf{n}:\alpha = \beta$ are called *basic nodes*. A basic node $\mathbf{n}: \alpha \parallel \gamma = \delta$ or $\mathbf{n}: \delta = \alpha \parallel \gamma$ *dominates* any node $\mathbf{n}':\alpha = \beta$ or $\mathbf{n}':\beta = \alpha$ which appears above $\mathbf{n}$ in the tableau in which $\beta \sqsubset \alpha$ and to which the rule REC has been applied.

---

[22]But without considering silent actions.

$$X_1 = a(X_1 \parallel X_4)$$
$$X_2 = aX_3$$
$$X_3 = (X_3 \parallel X_4) + bX_2$$
$$X_4 = b$$

Figure 19: An example family of BPP processes in standard form

REC $\qquad X_1 = X_2$

PREFIX $\qquad \overline{a(X_1 \parallel X_4) = aX_3}$

SUBL $\qquad \overline{(X_1 \parallel X_4) = X_3}$

REC $\qquad \overline{X_2 \parallel X_4 = X_3}$

SUM $\qquad \overline{a(X_3 \parallel X_4) + bX_2 = a(X_3 \parallel X_4) + bX_2}$

PREFIX $\qquad \overline{a(X_3 \parallel X_4) = a(X_3 \parallel X_4)} \qquad \overline{bX_2 = bX_2}$ PREFIX

$\qquad \overline{X_3 \parallel X_4 = X_3 \parallel X_4} \qquad \overline{X_2 = X_2}$

Figure 20: A successful tableau for $X_1 = X_2$.

SUBL, SUBR Whenever a basic node dominates a previous one, one of the SUB rules is applied to reduce the terms before applying the REC rule.

Theorem 27 *Every tableau for $\alpha = \beta$ is finite. Furthermore, the number of tableaux for $\alpha = \beta$ is finite.*

Proof (by contradiction): Let $T(\alpha = \beta)$ be a tableaux with root labelled $\alpha = \beta$, and assume that it is infinite. It can only be infinite if there exists an infinite path, since every node has only a finite number of possible branches, so let π be an infinite path starting from the root. Note that the only way in which π could be infinite is if it contains infinitely many applications of the REC rule. This is because the applications of the SUBL and SUBR rules will continually reduce the terms and due to the well-foundedness of $\sqsubset$ this process will eventually terminate. Thus π must contain an infinite sequence of basic nodes to which REC is applied. Let S be this sequence: $S = \{\mathbf{n}_i : \alpha_i = \beta_i\}_{i=1}^{\infty}$, where $\mathbf{n}_1 : \alpha_1 = \beta_1$ is the root, $\mathbf{n}_2 : \alpha_2 = \beta_2$ is the second node along π at which REC is applied, and so on. The contradiction will arise by considering π.

Since each expression α is $\in Var(\Delta)^{\otimes}$, it can be viewed as a vector $\overline{v}$ of $\mathcal{N}^n$, where the value of the ith coordinate of $\overline{v}$, denoted $\overline{v}(i)$, indicates the number of occurrences of variable X_i in α. Thus the sequence S can be represented by an infinite sequence of vectors $\{\overline{u}_i\}_{i=1}^{\infty}$ where $\overline{u}_i \in \mathcal{N}^{2n}$ for all i. The first n coordinates represent α_i and the last coordinates represent β_i.

Now the goal is to extract an infinite subsequence of S such that all coordinate sequences are nondecreasing. Consider first the infinite sequence $\{\overline{u}_i(1)\}_{i=1}^{\infty}$ consisting of all the first coordinates of vectors of the sequence S. If this sequence has an upper bound then extract from S an infinite sequence S_1 of vectors $\{\overline{v}_i\}_{i=1}^{\infty}$ with the property that the first coordinate of $\overline{v}_i$ remains constant throughout S_1. If the sequence $\{\overline{u}_i(1)\}_{i=1}^{\infty}$ does not have an upper bound then extract from S an infinite sequence S_1 of vectors $\{\overline{v}_i\}_{i=1}^{\infty}$ with the property that the first coordinate of $\overline{v}_i$ is nondecreasing. Continuing in this way for each coordinate of S results in an infinite sequence S_{2n} of vectors $\{\overline{w}_i\}_{i=1}^{\infty}$ with the property that all coordinate sequences are nondecreasing. Thus, in this

sequence every node is dominated by every node after it. Recall that a rule REC cannot be applied to a node if that node dominates a previous one, because either SUBL or SUBR must be applied first. This means that in S_{2n}, the rule REC cannot be applied to any node, thus resulting in the contradiction.

For the second claim of the theorem, the argument given by Christensen et al. is that since there are only a finite number of tableaux of a given finite size, then there can only be an infinite number of tableaux if there is some infinite sequence of partial tableaux (each derived from the previous one), which produces an infinite tableaux in contradiction to the first part of the theorem. The claim that there are only a finite number of tableaux for a given finite size seems to me to need some clarification. Although a minor point, "size" should be precisely defined; the number of rows is not adequate, since a row may have some (finite) number of entries on it, as the result of a SUM rule (e.g., see Figure 20). Perhaps "size" could be defined as "the number of $E = F$ expressions in a tableau" - e.g., the tableau in Figure 20 would have size 9. It would indeed follow that there can only be a finite number of tableaux for a given finite size, since the number of X_i is finite and so there cannot be an infinite number of tableaux for a given size k.

4.2.3   Completeness, Soundness, and Decidability

Theorem 28 (Completeness) *If $\alpha = \beta$ then there exists a successful tableau with root labelled $\alpha = \beta$.*

Proof: Suppose $\alpha = \beta$. If a tableau $T(\alpha = \beta)$ can be constructed with the property that any node $\mathbf{n} : E = F$ of T satisfies $E \sim F$, then by theorem (27) that construction must terminate, and so if the desired property indeed holds then each terminal will be successful and T will be a tableau for $\alpha = \beta$.

The desired property can be guaranteed if the rules of the tableau system can be shown to be *forward sound*, in the sense that if the antecedent as well as all nodes above relate bisimilar processes then the set of consequents relate bisimilar processes. This is straightforward from the properties of bisimulation and the definitions of the rules. For example, as mentioned above the rule REC is just an encoding of the expansion law for merge, and the forward soundness for SUBL, SUBR follow from the fact that bisimilarity is a congruence relation with respect to merge.

Remark: Christensen et al. define *forward soundness* as requiring that "if the antecedent...relate bisimilar processes then it is possible to find a set of consequents relating bisimilar processes." This seems unnecessarily weak, since the only rule that produces more than one consequent is SUM, and if the antecedent relates two bisimilar processes, then *all* the consequents must do so as well.

Theorem 29 (Soundness) *If there is a successful tableau for $\alpha = \beta$, then $\alpha \sim \beta$.*

Proof(by contradiction): Suppose $T(\alpha = \beta)$ is a tableau for $\alpha = \beta$, and that $\alpha \not\sim \beta$. A path $\pi = \{\mathbf{n}_i : E_i = F_i\}$ through T is constructed starting at the root in which $E_i \not\sim F_i$ for each i. Since the tableau must be finite, then π ends in a terminal node, $E_n = F_n$ for which $E_n \not\sim F_n$. By the very definitions of successful nodes and bisimularity, this means that such a terminal node cannot be successful, and so the tableau cannot in fact be successful.

The construction of π is very detailed and will not be presented in full here. The basic idea is that for any node $\mathbf{n}_i$ that relates processes that are not bisimilar, then it has a consequence $\mathbf{n}_{i+1}$ for which the same holds. It is shown how this is done for each of the rules, with SUBL and SUBR being the most difficult cases.

Theorem 30 *Bisimulation equivalence is decidable on BPP processes.*

Proof: Given some processes α, β, then from the two previous theorems, $\alpha \sim \beta$ iff there's some successful tableau with root $\alpha = \beta$. Since according to theorem (27), there are only a finite number of such tableau for $\alpha = \beta$, all that remains is to list systematically all such tableau and if a successful one is found, then $\alpha \sim \beta$. Note that it is important that the tableaux are listed systematically, since it can only be determined that $\alpha \not\sim \beta$ after all possible tableaux have been listed. Unfortunately, the Christensen et al. do not specify an algorithm for listing the tableaux, and such an algorithm can potentially be non-trivial.

4.2.4 Remarks

The communication operator included in BPP, $\|$, has no communication capabilities. The authors claim that the results can also be shown to hold if a limited form of communication, handshaking, is allowed. Thus BPP can be considered to be a subset of CCS in which all equations are guarded, there is no restriction (thus allowing the huge simplification of disregarding silent actions), and no relabelling. Interesting, [Baeten et al. 1993] leave it as an open question as to whether or not the bisimulation equivalence problem is decidable for PA. PA is distinguishable from BPP by the inclusion of left-merge and general, not just prefix, multiplication. It would be interesting to try to extend the methods of this proof to handle the PA system.

References

[Milner 1989] Calculus of Communicating Systems, Robin Milner.

[Baeten et al. 1993] Baeten, Bergstra, and Klop. Decidability of Bisimulation Equivalence for Processes Generating Context-Free Languages. *Journal of the ACM*. July 1993.

[Hopcraft & Ullman 1979] Hopcraft, J.E., and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[Christensen et al. 1993] Bisimulation Equivalence is Decidable for Basic Parallel Processes. *CONCUR '93*, pp. 143-157.

[Baeten and Weijland 1990] Process Algebra. J.C.M. Baeten & W.P. Weijland

[Bergstra & Klop 1985] Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science* 37, pp. 77-121.

[Bergstra & Klop 1984] Process Algebra for Synchronous Communication. *Information and Control* 60, pp.109-137.

[Christensen et al. 1992] Bisimulation Equivalence is Decidable for all Context-Free Processes. *CONCUR '92*, pp. 138-147.

[Hüttel & Stirling 1991] Actions Speak Louder than Words: Proving Bisimilarity for Context-Free Processes. *Proceedings of LICS 91*, pp. 376-386.